UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Aplicación Web de gestión de clubs de golf

Sergio Ruiz Aragón
Tutor: José Antonio Clavijo Blázquez
Ponente: Gonzalo Martínez Muñoz

Julio 2018

APLICACIÓN WEB DE GESTIÓN DE CLUBS DE GOLF

AUTOR: Sergio Ruiz Aragón
TUTOR: José Antonio Clavijo Blázquez

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2018

Resumen (castellano)

Este Trabajo Fin de Grado consiste en la realización de una aplicación web para gestionar clubs de golf. Debido a lo sofisticado del proyecto en sí, este solo abarca la parte del complejo cliente de la aplicación retomando el trabajo previamente realizado en la empresa, el cual fue el desarrollo de un complejo servidor y base de datos.

El proyecto procurará satisfacer los requisitos funcionales y no funcionales que se han acordado con el cliente de la aplicación web.

Esta aplicación web está enfocada principalmente a usuarios que son empleados de un club que tengan como tarea gestionar a los clientes, torneos o Green Fees del mismo, independientemente del cargo que desempeñen en el club.

Este proyecto no será diseñado y desarrollado para un solo club de golf, sino que se pretende que sea una aplicación genérica de gestión para su posterior distribución a diferentes clubs de golf.

La plataforma que mejor se adapta a la necesidad de tener que distribuir un producto a un amplio número de clientes es la web. En estos últimos años con el auge de la web como medio masivo de intercambio de información y como medio de comunicación social han cobrado mayor importancia las aplicaciones web, dejando de lado el desarrollo de aplicaciones de escritorio cuya distribución es más costosa.

No obstante, el objetivo principal de este proyecto es diseñar y desarrollar una interfaz usable, partiendo de las especificaciones y estándares formales y no formales de usabilidad web y la idea para llevar a cabo el proceso es la de buscar una tecnología o tecnologías adecuadas que faciliten dicha tarea.

Esta aplicación web se basa en la nueva librería de ReactJS, como principal aliado para realizar un diseño usable y mantenible en el tiempo, consiguiendo también así una aplicación más eficiente.

Abstract (English)

This Bachelor Thesis named *Web Application for managing golf clubs* consist in develop a managing tool to help golf club employees with their jobs. Because of the project complexity, this project will be focused on client-side operations starting with the previous server-side work. It will be attempted to meet all the functional and non-functional requirements agreed with the web application client.

This web application is fundamentally focused on golf clubs' employees who have the golf club managing task as main task, regardless of position. Administering golf club customers, members, Green Fees or tournaments are actions included in golf club managing task.

This project will not be designed and developed for just one golf club. The main intention is to develop and distribute a generic golf club managing web application for all the different golf clubs.

Web environment is the best way to reach all the potential clients. Nowadays, the exponential growth of the web, as well as the growth of the importance of social media and the exchange of information has caused the leaving of desktop applications, which distribution is more complex and expensive, and the expansion of web applications.

However, the main objective of this project is to design and develop an usable interface, following the formal and non-formal web usability standards and specifications. It will be carried out a research to identify the best technology, library or framework that fits better with this objective.

This golf club managing web application is developed in ReactJS. In this project, ReactJS is the best choice for making a maintainable, usable and efficient interface.

Palabras clave (castellano)

Golf, Green Fee,Tee time, Club de golf, React, JavaScript, Single Page Interface, Gestión, Aplicación Web, Usabilidad, Empleados.

Keywords (inglés)

Golf, Green Fee, Tee time, Golf Club, React, JavaScript, Single Page Interface, Managing, Web Application, Usability, Employees.

Agradecimientos

Este trabajo de fin de grado no habría sido posible sin la ayuda de Delonia Software S.L. quienes me han brindado la oportunidad y han depositado en mi la confianza de realizar este proyecto complejo y muy importante para esta entidad, partiendo de una base que ya se había estado construyendo.

Agradecer a los consejos de mis tutores formales y "no formales", compañeros, amigos, familia, que me han aconsejado y ayudado mucho en esta labor tan grande. Cada grano de arena suma.

No quería olvidarme de toda la gente que me ha enseñado a lo largo de la carrera, que siempre han estado entregados en su labor. Las noches en vela, y las largas horas delante de una pantalla y un teclado ahora dan sus frutos.

Un proyecto grande, pero que se ha hecho más pequeño y manejable gracias a todas esas personas. Mi más sincero agradecimiento.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Marco del proyecto

Este trabajo de fin de grado (TFG) trata sobre la realización del desarrollo completo de la parte cliente (coloquialmente llamado *Front-End)* sobre el desarrollo inicial de la lógica de negocio (coloquialmente llamado *Back-End*) y la base de datos realizados anteriormente en Delonia Software S.L.

Este proyecto permitirá a los diferentes usuarios del sistema acceder y gestionar cómodamente mediante una interfaz más intuitiva las diferentes tareas que requiere un club de golf. El objetivo principal que se busca con el desarrollo de esta interfaz de usuario es la de que los usuarios no tengan que formarse paralelamente en el uso de una aplicación específica, sino que con los conocimientos administrativos y de gestión que poseen les permitan usar la aplicación sin mayor complicación.

1.2 Motivación

Dado que la gestión de clubs de golf es un proceso relativamente complicado y que requiere de una formación específica que, en muchos casos, no se tiene o la empresa dedicada a este negocio en concreto debe destinar horas y recursos en formar a sus empleados en estas tecnologías. Esto conlleva la consiguiente pérdida de la productividad lo que afecta directamente a la actividad económica de dicha empresa.

Se toma como ejemplo en este ámbito de aplicaciones de gestión interna de una empresa el desfase que existe en torno al diseño de la interfaz de usuario y el poco enfoque que se hace a la experiencia de usuario. Estos son aspectos muy importantes y cabe decir que han cobrado una mayor importancia en estos últimos años como es el concepto de usabilidad en el ámbito de las TIC. Los usuarios de la aplicación deben ser capaces de aprender con una simple mirada dónde está la funcionalidad que en ese momento están buscando [1].

La usabilidad está centrada principalmente en el desarrollo de interfaces de usuario. Aunque este concepto se definió aún más en el año 2003 con Jakob Nielsen, el cual es uno de los gurús y expertos en usabilidad en plataformas web más conocido, este ya había ido cogiendo forma tiempo atrás con criterios como facilidad de uso, “amigable” con el usuario [2].

Aparte del poco enfoque a la usabilidad dentro del ámbito de aplicaciones internas de gestión en este caso, se ha podido observar que uno de los mayores problemas que existen es que en el 63% de los proyectos software superan el presupuesto estimado inicialmente [2] y es por ello por lo que destinar más recursos a realizar un mejor diseño, más simple, con tecnologías que facilitan el desarrollo de estas mismas, se deja atrás [2]. Es por esto último, que al realizar el diseño de esta aplicación me he querido centrar en el uso de nuevas tecnologías que, aparte de ser muy potentes, facilitan el aprendizaje de una manera mucho más rápida, ya que utilizan lenguajes de marcado comúnmente usados como puede ser HTML o de *scripting* como es JavaScript.

1.3 Objetivos

Los principales objetivos son:

- Crear una herramienta de gestión que facilite la labor de los empleados de un club de golf.
- Continuar con el desarrollo de un complejo proyecto previamente desarrollado en Delonia Software S.L.
- Crear una solución web enmarcada en el paradigma cliente – servidor.

Los objetivos secundarios de este proyecto son los siguientes:

- Documentarse en el uso de nuevas tecnologías para desarrollo dinámico de *front-end*.
- Realizar la integración del *front-end* con el *back-end* ya existente.

1.4 Organización de la memoria

La memoria consta de los siguientes capítulos:

1. **Introducción:** En este capítulo se verá un marco del proyecto que tratará sobre el contexto desde donde se parte y a quiénes va dirigida la aplicación. También se podrá comprender la motivación de este proyecto y los objetivos de este.
2. **Estado del arte:** En este capítulo se verá un poco la historia del golf. Tras ello se mostrarán las diferentes aplicaciones que existen de gestión de clubs de campos de golf y, por último, se hará alusión a las diferentes tecnologías para el desarrollo de aplicaciones web.
3. **Análisis y Diseño:** se verán los diferentes tipos de usuarios que existen en la aplicación y los requisitos funcionales y no funcionales de esta. También, en este capítulo, se verá brevemente algunos diagramas de flujo que se han realizado para las operaciones más complejas de la aplicación, así como decisiones de diseño. También se tratará de manera breve la idea de usabilidad.
4. **Desarrollo:** En este apartado se verán las diferentes tecnologías utilizadas en el proyecto y el porqué de su uso.
5. **Pruebas y resultados:** en este capítulo se hablará de manera breve las diferentes pruebas que se han hecho y se ejemplificará una de ellas.
6. **Conclusiones:** por último, se verán las conclusiones del proyecto y el trabajo futuro.

2 Estado del arte

2.1 Historia del golf

El golf nace a principios del siglo XV en alguna parte en la costa este de Escocia, aunque se sostiene la idea de que el juego puede que se iniciara en los Países Bajos. También existen indicadores de que ya en la época de los romanos se empezara a jugar a un juego muy similar a lo que hoy en día conocemos como golf, que consistía en el uso de un palo (Club) curvado y una especie de bola hecha de plumas. A pesar de todas estas aproximaciones, la primera referencia al término golf, tal y como lo conocemos hoy en día es en 1552 en la población de St. Andrews. No fue años más tarde cuando se empiezan a formar las primeras asociaciones de golf. Algunas de las más conocidas las podemos encontrar en Escocia como puede ser la denominada *Honourable Company of Edinburgh Golfers* cuyos inicios datan de 1744 y la mundialmente conocida y lo que se conoce hoy como "la cuna del golf" la *St. Andrews Society of Golfers* datando sus inicios en el año 1754. Más adelante, el golf se introduciría en Inglaterra en el año 1608. Concretamente en España, el primer club de golf se creó en 1891 en Las Palmas de Gran Canaria [3].

2.2 Gestión en un club de golf

Dado el éxito que ha tenido este deporte desde sus orígenes, la gestión de un club de golf ha sido una tarea de relevancia dado el volumen de usuarios. Dentro de un club de golf podemos encontrar las siguientes secciones que se encargan de gestionar partes en concreto de un club de golf:

1. **Gestión de instalaciones:** En un campo de golf este departamento se encarga de gestionar las diferentes infraestructuras, así como de su mantenimiento.

2. **Economía y finanzas:** Departamento encargado de la contabilidad, finanzas y asesoría económica de un club.

3. **Marketing:** Departamento encargado de comunicar ofertas de valor para los usuarios del club de golf, así como estudios de mercado y de identificar necesidades nuevas de los clientes.

4. **Gestión de empleados y recursos humanos:** Departamento encargada de la contratación, el buen cumplimiento de los objetivos por parte de sus trabajadores, el despido, la gestión de los contratos, pagos de impuestos, pagos a hacienda y seguridad social y asesoría a los propios empleados del club.

5. **Gestión operativa:** Departamento encargada de toda la actividad de un club, es decir, de gestionar y prestar el servicio adecuado y demandado a los clientes, así como gestionar material deportivo, torneos, trofeos, clasificaciones, etc.

2.3 Herramientas de gestión en el mercado

En este apartado vamos a analizar las diferentes herramientas de gestión que existen en el mercado para los diferentes clubs de golf. Como ya se comentaba anteriormente, existen dos grandes tipos de aplicaciones de gestión: aplicaciones de gestión a medida y aplicaciones genéricas de gestión. Existen numerosos ejemplos en el ámbito de aplicaciones internas en entornos empresariales, lo cuales no tienen cabida en nuestro ámbito de estudio, pero que, sin embargo, tienen en común algunas características como puede ser la complejidad de uso y la gran curva de aprendizaje que genera esto.

2.3.1 Aplicaciones de gestión genéricas

Dentro de las aplicaciones de gestión más genéricas, nos podemos encontrar principalmente con aplicaciones de gestión de reservas y citas que no están enfocadas únicamente en el mundo del golf, sino que también permiten realizar reservas para cualquier tipo de negocio como pueden ser citas médicas, reservas en restaurantes o centros de estética, spas entre otras opciones. Incluso si la plataforma no ofrece una plantilla prediseñada para los negocios que principalmente ha contemplado, en muchos ejemplos, el usuario puede crearse la suya propia, pero con un mayor coste para conseguir la funcionalidad completa de una plantilla ya prediseñada.

Un ejemplo concreto de este tipo de aplicación de gestión común es *Bookgy*. Su interfaz se puede ver en la figura 2-1. Esta permite una gestión simple y genérica de un campo de golf. Centrando la mirada en la interfaz gráfica, se puede observar que es una interfaz amigable. Podemos observar también que a simple vista se obtiene información muy clara de cada pantalla y solo la información que se requiere en ese momento. Vemos como la navegación por pantallas y menús de todo tipo (sliders, botonera de menú lateral) es intuitiva en cuanto a los movimientos que se deben hacer sobre ellos y la información que mostrarán al pulsar o moverse sobre los mismos. Pero, sin embargo, como ya se mencionaba anteriormente, las opciones son algo reducidas y solo se permiten reservas en campos específicos de 9 y 18 hoyos dejando opciones de 27, 36 y 72 hoyos fuera del ámbito de la aplicación como son por ejemplo los campos en “Resorts”. Aunque esta aplicación disponga de una plantilla prediseñada para servir a clubs de golf, vemos que gran parte de su diseño está reciclado de otras interfaces o plantillas para otros negocios [4].

Figura 2-1: Interfaz de usuario de Bookgy.
Fuente: https://bookgy.com/es/

Por otro lado, tenemos la aplicación de gestión *Concept Golf*, una aplicación mucho más completa que la anterior, centrada en el negocio del golf. El producto ha sido desarrollado teniendo en cuenta la flexibilidad necesaria para gestionar el sistema dependiente de los requerimientos de cada campo de golf; desde campos en Resorts de 72 hoyos para pagar y jugar, hasta campos de socios de 18 hoyos. Con más de 150 informes, este sistema es esencial para gestionar cualquier campo de golf [5]. Se puede observar como la interfaz gráfica de esta aplicación es muy poco amigable, como se puede ver en la figura 2-2. A simple vista no podemos saber dónde se encuentra toda la funcionalidad del producto, y no parece ser nada intuitiva. No existe, como se puede apreciar en la figura 2-2, un orden en los formularios mostrados en la pantalla y la pantalla no se muestra muy atractiva e incluso se podría decir que genera rechazo meterse a rellenar un formulario o tocar cualquier botón, puesto puede que se llegue a algún estado o un caso de uso no deseado [5]. Esto hace que, para usar este software, se requiera de un conocimiento previo amplio o abarcando los casos de uso que comúnmente suelen aparecer en el uso diario de la aplicación.

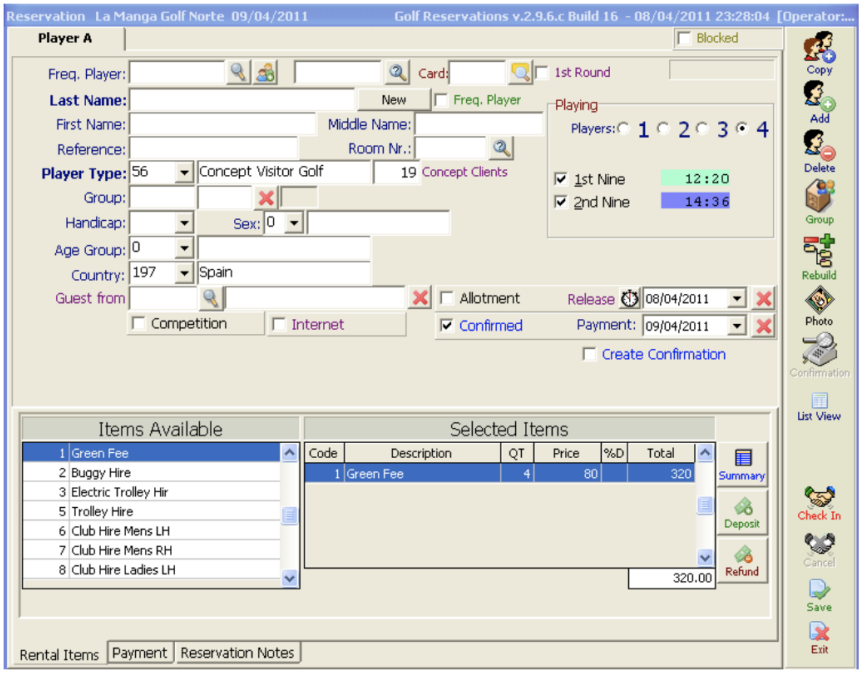

Figura 2-2: Interfaz de usuario, pantalla de reservas, de Concept Golf.
Fuente: http://www.csscorporate.com/en/products/concept-golf-management-software/

2.3.2 Aplicaciones de gestión no genéricas

Una gran ventaja de las aplicaciones no genéricas es que están hechas a medida para el cliente final, optimizando así sus recursos y centrándose en los flujos principales y el funcionamiento específico de cada negocio en concreto. Una de las principales pegas de este tipo de aplicaciones, es que los presupuestos son algo más elevados que las aplicaciones estándar o comunes. Con ello, una gran cantidad de empresas demandantes de este tipo de aplicaciones en concreto, suelen ahorrarse los costes y el tiempo extra de diseño que se suele dedicar para plantear una aplicación que cumpla con, al menos, algunos estándares de usabilidad y por ello sacrificar la calidad del producto. En muchas ocasiones, esto es debido a que la empresa encargada del desarrollo del proyecto no tiene

muy claro cuáles son estos estándares de usabilidad, los cuales serán nombrados en el punto 4 de este documento; o, por el contrario, el equipo de desarrollo es conocedor de estos estándares, pero no conoce el uso de tecnologías y herramientas de desarrollo más ágiles y orientadas a satisfacer estas necesidades en el producto final [6].

Al ser una aplicación a medida, se garantiza que el uso será privado para la empresa cliente del producto. A su vez, se le suele ofrecer al cliente dentro del presupuesto del producto, el precio de un contrato de mantenimiento (generalmente reducido) que asegura la continuidad de este en un periodo de tiempo establecido. Esto asegura que el sistema reciba continuas actualizaciones y mejoras a favor de que la productividad de los empleados de gestión del campo de golf se vea aumentada [6].

2.4 Tecnologías web disponibles

Vistos los diferentes tipos de productos que existen en el ámbito de las aplicaciones de gestión de clubs de golf, se expondrán las diferentes tecnologías web disponibles para el desarrollo de este tipo de productos. A pesar de la complejidad de este proyecto, estas herramientas facilitarán el desarrollo de la aplicación web.

2.4.1 HTML

El significado de estas siglas es el de Lenguaje de Marcado para Hipertextos o en inglés *Hypertext Markup Language* (HTML). Es el medio de construcción de páginas web más básico y se usa principalmente para ordenar y representar de manera visual los elementos que conforman una página web. Pero simplemente representan de manera visual una página web, pero no aporta ninguna funcionalidad de ningún tipo a los diferentes elementos de esta. Sin embargo, HTML ha sufrido cambios a lo largo de la historia y se ha podido observar cómo se ha ido adaptando a las nuevas necesidades dentro de la informática [7][8]. Con las nuevas necesidades en estos últimos tiempos del mayor intercambio de información de una manera rápida y sencilla, así como con el nacimiento de la idea de usabilidad, nace el término acuñado por Tim O'Reilly llamado *Web 2.0* allá por el año 2004 [9]. Este concepto se refiere a que páginas web atienden a ideas como la usabilidad, interoperabilidad; conceptos que se relacionan en mucha medida con la idea de que es el usuario el protagonista e incluso con la idea de que el usuario genera el propio contenido de la página web, como se puede observar en las redes sociales, ya que es el usuario final el que interactúa con la página web publicando contenido propio diverso como por ejemplo contenido multimedia, texto, etc. [9]. Retomando los diferentes cambios que se han ido realizando sobre HTML a lo largo de su vida, las ideas que van detrás del concepto de Web 2.0, han hecho que el propio HTML sea capaz de dar funcionalidad a los elementos que se declaran en un documento HTML si este es interpretado por un navegador. Con ello se consigue que ideas como usabilidad sean más fácil de implementar en las nuevas versiones de HTML. Actualmente se encuentra en su versión 5.2, la cual facilita la organización del documento con nuevas etiquetas disponibles para el usuario y con la inclusión de nueva funcionalidad como la validación de valores en entradas de datos, manejo de audio y video, etc. [10].

2.4.2 CSS

El significado de estas siglas es el de Hojas de Estilo en Cascada o en inglés *Cascading Style Sheets* (CSS). Podemos decir que CSS es el lenguaje que se utiliza para dar forma y diseño a los documentos HTML, SVG o XML. Este es el aspecto más importante a la hora de presentar un documento al usuario, ya que la primera información que le va a entrar va a ser a través de los ojos, mucho antes de empezar a analizar y detenerse en el contenido [11][12].

Es por ello, que junto con las herramientas de organización de elementos que se brinda con HTML y junto con CSS, hacen que diseñar una página web más usable sea una tarea mucho más fácil. A parte, con la nueva versión de CSS, llamada CSS3, se ha ayudado a llevar a cabo esta tarea de una manera mucho más cómoda debido a la inclusión de nueva funcionalidad en el lenguaje como es la inclusión de nuevos *layouts* o el manejo de ciertas propiedades de manera más cómoda, creación de animaciones, transiciones, la inclusión de *flexboxes* para manejar de una manera mucho más eficiente la posición o la colocación de elementos dentro de un documento HTML [12].

2.4.3 JavaScript

JavaScript es, por excelencia, el lenguaje interpretado más usado para brindar de funcionalidad a las páginas web y a cada uno de sus elementos. Técnicamente hablando, la principal función de JavaScript es manejar el Modelo de Objetos del Documento o en inglés *Document Object Model* (DOM) en paradigmas servidor-cliente. Esta siempre ha sido la visión "clásica" de dicho paradigma, donde el cliente tenía una menor importancia debido a su menor capacidad de computación comparado con los servidores. Hoy en día, se ha dado la vuelta al paradigma servidor-cliente para dar paso al paradigma cliente-servidor, donde los terminales cliente poseen una mayor potencia de computación y se les delega una mayor cantidad de tareas [13]. Sin embargo, JavaScript no solo se queda ahí. Clasificar JavaScript es bastante difícil. Se podría decir que JavaScript es un lenguaje destinado al scripting en páginas web, pero esto sería limitar la vista a solo una parte de su funcionalidad total. JavaScript no solo aporta funcionalidad y dinamismo a páginas web, sino que también se utiliza en muchos entornos sin navegador. Vemos que con Node.js, su uso se ha extendido al *back-end* o lado del servidor [14].

Se puede decir que este es uno de los lenguajes más versátiles que se conocen en la actualidad ya que no solo se centra en la parte de scripting, sino que soporta estilos de programación funcional, orientada a objetos e imperativa [14].

Su uso se ha popularizado mucho en estos últimos años, desde que, en 2012, todos los navegadores soportan la versión de JavaScript ECMAScript 5.1; pero no es en 2015 cuando ECMA Internacional lanza la sexta versión de JavaScript ECMAScript que recibe el nombre de JavaScript ECMAScript 2015. A partir de entonces, cada año se lanzan versiones anuales [14].

Gracias a su popularidad, se han desarrollado varios *frameworks* sobre JavaScript muy conocidos y usados en la actualidad. Alguno de estos son *ReactJS* [15], *AngularJS* [16]*, o Vue.js* [17] entre otros.

En este caso se ha elegido *ReactJS* para desarrollar el *front-end* de este proyecto debido al modularidad que aporta y la gran velocidad de renderizado que es capaz de conseguir debido al uso de un DOM Virtual del que ya hablaremos más adelante.

2.4.4 Java + GWT

Por excelencia, Java es uno de los lenguajes más usados y conocidos y es por ello por lo que su uso no solo se ha limitado al desarrollo de la parte del servidor o *back-end*, sino que han aparecido muchos *frameworks*, librerías, generadores de código, compiladores, preprocesadores entre otros, para el desarrollo de interfaces gráficas de usuario.

Si bien es cierto que su uso es algo menos común que el que puede ser de HTML, CSS y JavaScript; aún muchas grandes empresas siguen usando Java para desarrollar interfaces gráficas.

Uno de estos generadores, compiladores y preprocesadores es GWT [19][20]. Esta herramienta es capaz de generar código JavaScript y HTML a partir de código Java. El desarrollo en Java es muy parecido a librerías de desarrollo de interfaces gráficas para aplicaciones de escritorio como es Java Swing [18].

Aparte de todo esto, *Java GWT* combina muy bien con el lado del servidor pudiendo utilizar los mismos objetos Java en el lado del servidor o en el lado del cliente. Posee a su vez herramientas de depuración sobre el código Java y no sobre el código autogenerado JavaScript, con lo cual facilita mucho la localización de errores en tiempo de ejecución [19][20].

2.4.5 PHP

Por último, dentro de esta presentación de las diferentes tecnologías web, no se puede dejar fuera a uno de los primeros lenguajes diseñado para el desarrollo web de contenido dinámico como es PHP o en inglés conocido como *PHP Hypertext Preprocessor* [21][22].

La ventaja del código PHP es que se puede incrustar en el propio HTML y así conseguir dinamismo en las páginas web. El código PHP debe ser interpretado y ejecutado en el lado del servidor para después poder generar una respuesta al cliente en forma de documento HTML. El requisito fundamental es que dicho servidor web posea un módulo procesador de PHP que se encargará de realizar las tareas comentadas en este párrafo [21][22].

Se considera un lenguaje fácil de aprender ya que está basado en la sintaxis y la estructura de lenguajes de paradigma imperativos como puede ser C. Aunque el código PHP se pueda embeber en un documento HMTL, este es invisible para el navegador web, ya que el servidor se encarga de interpretar y ejecutar ese código PHP y de enviar una respuesta, que será un documento HTML plano [22].

El inconveniente más notable que posee PHP es que se interpreta en el servidor en tiempo de ejecución. Esto plantea dos problemas principales: en primer lugar, los errores solo se descubren en tiempo de ejecución y si se ejecutan ciertas líneas de código que generen un problema, quedando ocultos otros errores que se podrían solventar en tiempo de

compilación. En segundo lugar, los mensajes de error se muestran en el HTML resultante, con lo que se pueden ver directamente impresos por pantalla, mostrando demasiada información acerca del código fuente [21][22].

3 Análisis y Diseño

3.1 Análisis del problema

El problema planteado por parte del cliente de la aplicación de gestión de campos de golf es la de ser capaces de crear una aplicación de gestión a medida. Dado que este proyecto se compagina con un *back-end*, el cual no entra dentro del ámbito de este proyecto, aparecen requisitos funcionales y no funcionales de ambas partes, ya que, en mayor o menor medida, afectan al diseño y posterior desarrollo de este complejo proyecto. También se expondrá una breve descripción de los diferentes tipos de usuarios que existen en la aplicación. Antes de comenzar, es conveniente recordar que tipos de requisitos existen.

- Requisito funcional: Se comprende como aquellas descripciones de la funcionalidad o servicios que debe ofrecer un sistema, el comportamiento en determinados contextos y las salidas a diferentes entradas al sistema [26].
- Requisito no funcional: Se comprende como aquellas condiciones que se imponen al sistema en general como pueden ser: condiciones de tiempo, de velocidad, etc. [27].

3.1.1 Tipos de usuarios en la aplicación

A continuación, se expondrán los diferentes tipos de usuarios que pueden existir en la aplicación, como se puede ver en la tabla 4-1. Cada uno de ellos tiene ciertos roles en la aplicación y, por lo tanto, poseerán diferentes privilegios a la hora de usar la misma. Esto, traducido al dominio de este proyecto, consistirá en que a cada uno de estos usuarios se les mostrará un tipo de información diferente en las pantallas que así lo requieran. De este modo se establece también cierta seguridad en la aplicación, debido a que se protegen ciertos datos que no pueden estar en manos de usuarios incorrectos, así como la posible eliminación o manipulación de estos.

Usuario	Rol
Administrador Global	Usuario con permiso total sobre la aplicación. Si la empresa posee varios campos de golf, podrá administrarlos todos.
Administrador del Club	Usuario con permiso total sobre el club al que pertenece.
Empleado	Usuario estándar de la aplicación el cuál hará un mayor uso de este. Permisos para gestionar a los clientes, las reservas del club, torneos
Profesor	Usuario con acceso a secciones y datos que se relacionen con él. Como por ejemplo datos de alumnos.
Organizador de torneos	Usuario con acceso a datos relacionados con los torneos del club

Tabla 3-1: Roles de usuario

Aunque en los requisitos funcionales basados en las historias de usuario, en este caso de nuestro cliente, no se especifica que los usuarios se dividan en dos grandes grupos, se

puede observar como en la tabla 4-1 existen dos posibles agrupaciones de usuarios: administradores y usuarios estándar. Estos usuarios no dejan de ser en "la vida real" trabajadores del club de golf donde pueda estar funcionando este software en un futuro; pero es necesario que existan roles y permisos en la propia aplicación como se comentaba en el párrafo anterior. A continuación, se verá con detalle cada usuario y su correspondiente rol.

3.1.2 Requisitos funcionales

A continuación, se expondrán los requisitos funcionales acordados entre la empresa y el cliente. El sistema, y con ello los requisitos funcionales, están divididos por subsistemas con el fin de organizar el trabajo de una manera más eficiente y ordenada. En cada subsistema se trata de abarcar funcionalidad común, como puede ser, por ejemplo, el subsistema de gestión de usuarios. A la hora de diseñar y desarrollar la interfaz y sus componentes, no se mezclará otro tipo de funcionalidad durante estas fases y será posible interactuar con el cliente. Así, se podrá trabajar mediante hitos con entregas parciales del proyecto al cliente para que este verifique el cumplimiento de los requisitos propuestos.

3.1.2.1 Subsistema de gestión de usuarios

RF1. Cualquier usuario registrado podrá iniciar sesión en la aplicación.

RF2. Cualquier usuario podrá recuperar su contraseña, y este recibirá un email con las instrucciones de recuperación y un código.

RF3. Cualquier usuario que haya recibido el email de recuperación de contraseña, introducida el código de recuperación y cambiara su contraseña.

RF4. El administrador global podrá administrar a cualquier usuario del sistema.

RF5. El administrador del club podrá administrar a los usuarios pertenecientes a su club.

RF6. El empleado podrá registrar profesores y organizadores de torneos en la aplicación.

RF7. Cualquier usuario de la aplicación puede cambiar su contraseña.

3.1.2.2 Subsistema de gestión de clientes y socios

RF8. El empleado podrá administrar los clientes del club (Permitiendo buscar por nombre, DNI, edad, si es socio, por tipo de socio, estado de las cuotas, sexo, provincia, nacionalidad).

RF9. El empleado puede administrar los clientes que sean socios del club sin exceder el límite de clientes en la membresía.

RF10. El empleado puede dar de alta una membresía y se notificará al cliente vía SMS o email con los detalles de su membresía.

RF11. El empleado puede ver el detalle del cliente.

RF12. El empleado puede ver el detalle de la suscripción de un socio.

RF13. El empleado podrá ver el listado de las reservas del cliente (Permitiendo buscar por fecha).

RF14. El empleado podrá ver el detalle de cada una de las reservas del cliente.

RF15. El empleado podrá cambiar en cualquier momento los permisos de envío de SMS y email del cliente por petición de este.

RF16. El empleado podrá ver todas las comunicaciones del cliente por SMS y email (Permitiendo buscar por fecha y tipo de comunicación).

RF17. El empleado podrá administrar las alertas de un cliente (Permitiendo buscar por tipo de alerta y fecha).

RF18. El empleado podrá marcar una alerta como "No mostrar más" o "Tratada".

RF19. El empleado podrá marcar como bloqueado o desbloqueado a un cliente.

RF20. El empleado podrá subir fichero para adjuntarlos al cliente.

RF21. El empleado podrá ver un listado con los jugadores que juega habitualmente un cliente (Permitiendo buscar por nombre y número de veces jugado).

RF22. El empleado podrá administrar los tipos de socios.

RF23. El empleado podrá administrar las formas de pago de los tipos de socios.

RF24. El empleado puede administrar las direcciones del cliente.

3.1.2.3 Subsistema de gestión de campos y productos

RF25. Un administrador de club podrá administrar los recorridos del club.

RF26. Un administrador de club podrá administrar los hoyos del club.

RF27. Un empleado podrá administrar los artículos.

RF 28. Un empleado podrá administrar las instancias de los artículos.

RF29. Un empleado podrá administrar las sesiones de cada recorrido.

RF30. Un empleado podrá administrar los precios de las reservas de las sesiones, dependiendo del tipo de socio, del canal de reserva, de si es individual o en grupo.

RF31. El empleado podrá crear ofertas puntuales, promociones u ofertas de último minuto.

RF32. El empleado podrá administrar packs con los Green Fees y los diferentes productos.

RF33. El empleado podrá bloquear la instancia de un artículo por un tiempo o indefinidamente, o desbloquearlo.

RF34. El empleado podrá establecer los días de cierre del club.

RF35. El empleado podrá establecer el estado de un recorrido y el periodo que va a durar ese estado.

RF36. El administrador del club puede añadir una foto de perfil del club.

RF37. El empleado puede adjuntar fotos a un recorrido.

3.1.2.4 Subsistema de gestión de reservas

RF38. Un empleado podrá realizar una reserva eligiendo: Fecha de la reserva, Tee time, los extras (opcionales), los packs (opcionales), el número de jugadores, datos de los jugadores, el recorrido, el profesor con el que quiere dar la clase (opcional).

RF39. Un empleado podrá asignar una instancia de un artículo a una reserva, cuando el cliente va a hacer efectiva la reserva en el club.

RF 40. En el caso de que la persona que hace la reserva no esté registrada en el sistema, un empleado podrá darla de alta como cliente.

RF 41. Un empleado podrá modificar el Tee time de una reserva, siempre que no esté pagada o el precio no cambie al mover la reserva.

RF42. Un empleado podrá modificar los jugadores o extras de una reserva, siempre que no esté pagada o el precio no cambie al mover la reserva.

RF43. Un empleado podrá cancelar una reserva.

RF44. Un empleado podrá ver un listado de todas las reservas.

RF 45. Un empleado podrá ver los datos, jugadores y extras de una reserva.

RF46. Un empleado podrá darle a un botón llamado "Que me espera hoy", el cual mostrará un listado con todas las reservas para el día actual y sus respectivos jugadores.

RF 47. Un empleado podrá ver las tarifas más populares.

RF48. Un empleado podrá hacer comparativas de los ingresos de distintos días.

RF49. Un empleado podrá marcar una reserva como pagada.

RF50. Un empleado podrá marcar una reserva como utilizada.

RF51. Un empleado podrá indicar en un precio de sesión el importe a pagar si se cancela la reserva.

RF52. Un empleado podrá generar los Tee time de un recorrido, indicando un periodo, el número máximo de jugadores y la diferencia de tiempo entre ellos.

RF53: Un empleado podrá gestionar los abonos para el club donde trabaja y dar de alta a nuevos clientes en el caso de que sean nuevos.

3.1.2.5 Subsistema de gestión de torneos

RF54. Un empleado podrá administrar los torneos.

RF55. Un organizador de torneos podrá administrar los torneos.

RF56. Un empleado podrá administrar los jugadores de torneos.

RF57. Un organizador de torneos podrá administrar los jugadores de torneos.

RF58. Un empleado podrá administrar los registros al torneo.

RF59. Un organizador de torneos podrá administrar los registros al torneo.

RF60. Un empleado podrá introducir los resultados del torneo de cada jugador.

RF61. Un organizador de torneos podrá introducir los resultados del torneo de cada jugador.

RF62. Un empleado podrá imprimir las etiquetas con el nombre, licencia y hándicap para pegarlas a las tarjetas de juego.

RF63. Un organizador de torneos podrá imprimir las etiquetas con el nombre, licencia y hándicap para pegarlas a las tarjetas de juego.

RF64. Un organizador de torneos podrá enviar el resultado del torneo a la Federación Española de Golf mediante el servicio web provisto por ellos.

3.1.2.6 Subsistema de gestión de profesores

RF65. Un profesor podrá ver su agenda y posibilidad de reserva.

RF66. Un profesor podrá administrar sus horarios de disponibilidad.

RF67. Un profesor podrá administrar los precios en cada horario, dependiendo del día, la duración de la clase, categoría de la clase (individual o para grupos), para socios o no socios.

RF68. Un empleado podrá administrar los profesores.

RF69. El cliente y el profesor recibirán un email de confirmación donde se les informará de los detalles de la reserva.

RF70. Un profesor podrá administrar sus idiomas.

RF71. Un empleado podrá administrar los idiomas de un profesor. Un empleado podrá bloquear o desbloquear a un profesor.

3.1.2.7 Subsistema de gestión de facturas

RF72. Un empleado podrá ver los detalles que van a incluirse en el fichero de domiciliación bancaria que se envía al banco con las cuotas de socio.

RF73. Un empleado podrá exportar el fichero anterior a Excel.

RF74. Un empleado podrá volver a lanzar un recibo no cobrado o rechazado.

RF75. Un empleado podrá notificar al cliente por una cuota no pagada.

RF76. Un empleado podrá generar una carta (PDF, basado en una plantilla predefinida) con los datos de la cuota del cliente.

RF77. Un empleado podrá ver un listado con las cuotas del cliente. Un empleado podrá marcar una cuota como pagada.

RF78. Un empleado podrá generar facturas (PDF, basado en una plantilla predefinida) para los clientes que lo deseen.

RF79. Un profesor podrá generar los datos suficientes para generar su factura.

RF80. Un administrador global podrá generar facturas (PDF, basado en una plantilla predefinida) de cada club.

RF81. Un administrador global podrá administrar clubs.

RF82. Un administrador global podrá bloquear o desbloquear a un club.

RF83. Un administrador global podrá ver lo ingresos generados en un rango de tiempo.

RF84. Un administrador global podrá ver la afluencia de clientes en rangos de tiempo.

RF85. Un administrador de club podrá administrar el IVA de su club.

3.1.2.8 Subsistema de gestión de clubs

RF86. Un administrador global podrá administrar clubs.

RF87. Un administrador global podrá bloquear o desbloquear a un club.

RF88. Un administrador global podrá ver lo ingresos generados en un rango de tiempo.

RF89. Un administrador global podrá ver la afluencia de clientes en rangos de tiempo.

3.1.3 Requisitos no funcionales

3.1.3.1 Rendimiento

RNF1. La carga y renderizado de la página se debe cumplir en menos de 15 segundos.

3.1.3.2 Portabilidad

RNF2. El sistema debe poder funcionar en navegadores más antiguos (Internet Explorer 10, Firefox 24.0, Chrome 29.0 y Safari 5; en adelante)

3.1.3.3 Usabilidad

RNF3. Debe tener una interfaz intuitiva, para que un nuevo empleado no tarde más de un día en aprender a utilizar el sistema.

RNF4. Se debe acceder a los procesos de nueva reserva o hacer efectiva una reserva en menos de dos clics.

3.2 Diseño de la solución

Tras realizar el correspondiente análisis del software, se procederá a realizar el correspondiente diseño. Un buen diseño asegura que el software sea robusto, mantenible, eficiente, limpio y ordenado, así como un software de calidad.
.

3.2.1 Usabilidad

Antes de hablar de usabilidad, se hablará de la importancia del contenido, ya sea en una página web o en una aplicación que se va a ofrecer posteriormente a unos usuarios.

Principalmente, el contenido es aquello que se provee a los usuarios en un sitio Web. Pero, sin embargo, en muchas ocasiones, las páginas web centran todos sus recursos en proveer fácil acceso y una buena usabilidad a contenido innecesario, erróneo, o carente de significado para el usuario. Algunos estudios revelan que el contenido es mucho más importante que la navegación en la página web, el diseño visual, la funcionalidad y la interactividad [23].

Cabe decir que multitud de organismos gubernamentales desarrollan estándares y recomendaciones de usabilidad entre otros temas, muchas veces respaldado por la *World Wide Web Consortium* (W3C) [25]. Es por ello por lo que, para ejemplificar este hecho, se ha querido obtener la definición de usabilidad propuesta en este apartado, desde el punto de vista de un organismo gubernamental.

Según la definición propuesta por la Guía Web con la normativa actualizada para Sitios Web del Gobierno de Chile, División de Gobierno Digital, se puede entender la usabilidad como la medida de la calidad de la experiencia que tiene un usuario cuando interactúa con un producto o sistema. Esto se mide a través del estudio de la relación que se produce entre las herramientas (entendidas en un sitio web el conjunto integrado por el sistema de navegación, las funcionalidades y los contenidos ofrecidos) y quienes las utilizan, para determinar la eficiencia en el uso de los diferentes elementos ofrecidos en las pantallas y la efectividad en el cumplimiento de las tareas que se pueden llevar a cabo a través de ellas [24].

Los estándares de usabilidad están recogidos por la organización internacional de estandarización ISO. Son varios los documentos que recogen los estándares formales de usabilidad, pero en esta sección nos centraremos en aquellos que proponen una definición de usabilidad, así como de algunas características relacionadas con la misma.

En primer lugar, se mencionará la definición propuesta por la ISO 9241-11:1998 "Ergonomics of Human System Interaction":

"La medida con la que un producto se puede usar por usuarios determinados para conseguir objetivos específicos con efectividad, eficiencia y satisfacción en un contexto de uso concreto" [32].

Por otro lado, la ISO/IEC 9126-1:2001 expone un modelo de calidad basado en unas características que todo software de calidad debe cumplir. Estas son: funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad [33].

De todas estas características que se proponen en este documento, se va a describir las diferentes subcaracterísticas recogidas en la usabilidad.

- Fácil de aprender: capacidad para ser aprendido en su uso [33].
- Fácil de entender: capacidad del producto de poder ser analizado y descubrir cómo puede ser usado en tareas específicas [33].
- Fácil de operar: capacidad del producto software para ofrecer comodidad a la hora de realizar tareas y ejecutar acciones [33].
- Atractivo: forma en la que el producto software despierta interés en el usuario e invita a su uso [33].
- Conformidad: capacidad de un producto software de adherirse a un estándar, convención, guía de estilo o regulaciones relacionadas con la usabilidad [33].

Estas características relacionadas con la usabilidad han hecho que se tomen ciertas decisiones de diseño para simplificar una web compleja con bastante funcionalidad que manejar.

3.2.2 Diagramas de flujo de la aplicación

Un diagrama de flujo es una herramienta necesaria para la representación de un algoritmo, o proceso específico dentro del programa.

En este proyecto se han realizado algunos diagramas de flujo junto al cliente con el fin de establecer y aclarar algunos conceptos y funcionalidad primordial en la aplicación.

Se ha considerado que los principales procesos de la aplicación son las que realiza el empleado del campo de golf y son las siguientes:

- Generar una nueva reserva.
- Gestionar una reserva efectiva, es decir, realizada con anterioridad por un cliente.
- Cancelación de una reserva.
- Y dar de alta a un nuevo miembro del club (membresía) sea cliente previo o no.

Los diagramas de flujo de todos estos procesos se ven reflejados en el Anexo B de este documento.

3.2.3 Decisiones de diseño

Dado que se trata de un sistema bastante complejo, se ha decido buscar un diseño que facilite la tarea de implementar una interfaz que englobe toda la funcionalidad requerida, a la par que se atienden a algunas recomendaciones de usabilidad comentadas en el punto 3.2.2. En una primera instancia se pensó en un sistema de navegación clásico por pantallas. Es decir, plantear un menú lateral con todas las opciones en entradas de menú y submenús, recargando la página en su totalidad, con la consiguiente pérdida de rendimiento.

La opción de diseño más acorde con la complejidad del sistema ha sido centrarse en el patrón de diseño ***Single Page Interface*** (SPI) [28].

En una aplicación web basada en SPI funciona a partir de estados (su nombre habitual es *States*) [28]. La idea principal es que toda la página web se carga al principio, en la primera llamada al servidor y luego se va cargando de forma dinámica sin la necesidad de recargar la página entera [29]. En SPI se mezcla también la idea del paradigma cliente-servidor donde el cliente tiene mucho mayor peso que el servidor, al contrario que en el paradigma servidor-cliente. Es lo que se conoce también como "Arquitectura de Servidor Delegado" [29]. Se delega en el cliente todo el procesamiento de la aplicación web sin tener que realizar más llamadas al servidor salvo que se necesita algún dato que se encuentre almacenado en base de datos. El inconveniente más grande es que la primera carga es más lenta de lo normal, ya que el cliente tiene que descargar todos los módulos de la aplicación web, así como los primeros datos para la pantalla que está cargando. Esta idea se desarrollará más en el apartado de desarrollo de este documento [29]. En la figura 3-1 se puede apreciar el patrón de diseño propuesto. Se puede ver como toda la lógica de la interfaz se maneja del lado del cliente bajo módulos escritos en JavaScript (figura 3-1 esquina superior izquierda). Los datos de sesión y variables, así como algún conjunto de datos, se almacenan en el almacenamiento local, o lo que se conoce comúnmente como *local storage*. Por último, se puede apreciar como la comunicación con el servidor en este patrón requiere que sea mediante *eXtensible Markup Language* (XML) [30] o mediante *JavaScript Object Notation* (JSON) [31]. Con esto también se cumplen algunas de las características de usabilidad como por ejemplo la facilidad de operar y de entender. Para el usuario es más fácil ver cómo cambian componentes pequeños, que ver como se refresca la página entera con el consiguiente de que puede llegar a perder datos de una pantalla a otra o de perderse en la navegación. Al usuario también le genera disconformidad el ver que la navegación no es fluida [23].

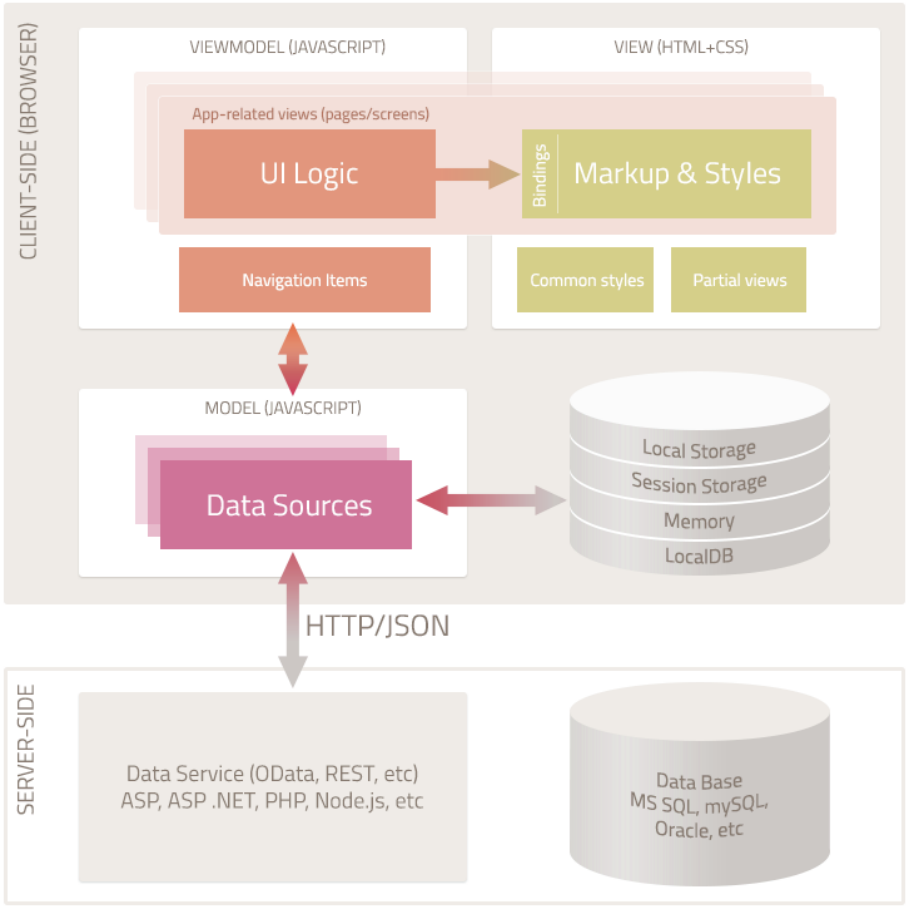

Figura 3-1: Esquema del patrón de diseño Single Page Application. Fuente: https://upload.wikimedia.org/wikipedia/commons/1/1b/Single_Page_Applications_structure.png

4 Desarrollo

4.1 ReactJS

Dado que se ha decidido partir de un diseño basado en el patrón *Single Page Interface* y se pretende hacer uso de una arquitectura *front-end* basada en HTML, CSS y JavaScript, se ha decidido usar en este proyecto ReactJS. ReactJS es una librería sobre JavaScript que permite crear interfaces de usuario de una manera sencilla y eficiente.

La diferencia principal con otras herramientas de desarrollo web es que ya no se delega gran parte del procesamiento de la aplicación web en el servidor (*back-end*) sino que gracias a ReactJS y a su DOM Virtual, que optimiza los recursos y mejora la eficiencia, se puede realizar más trabajo en el lado del cliente, como algunos cálculos de datos, validaciones complejas de campos entre otras soluciones; lo que hace que esta librería encaje a la perfección con el patrón de diseño elegido.

Su DOM virtual es una solución más avanzada que el propio DOM que ofrece el navegador. El DOM es una interfaz de plataforma que define la manera en que objetos y elementos se relacionan entre sí en el navegador [34]. El DOM virtual comprueba los cambios que ha habido entre objetos de la versión nueva y la almacenada previamente en el DOM Virtual. Así pues, se consigue que únicamente se actualice las partes necesarias del DOM del navegador.

Se explicará brevemente el funcionamiento de ReactJS a efectos teóricos y prácticos para justificar el uso de esta librería.

4.1.1 Componentes de ReactJS y su ciclo de vida

La idea principal de ReactJS y por la que se ha escogido para este proyecto, es por su capacidad de dividir el problema en pequeños subproblemas. Es por ello por lo que ReactJS tiende a que cada elemento visual complejo esté formado por pequeños elementos visuales más simples llamados componentes. Así, se consigue que el código quede modularizado de por sí, con el consiguiente de que se obtiene una mayor reutilización del código y una mejor mantenibilidad para el futuro, así como lograr cuidar cada componente y hacer que la tarea de realizar una web usable sea más sencilla. Algunos puntos de usabilidad como la velocidad de renderizado y carga de la página, así como por ejemplo la creación de validaciones propias de la aplicación web y de elementos emergentes propios, se han podido incluir de manera más sencilla en este proyecto gracias a ReactJS.

ReactJS junto con las nuevas versiones de JavaScript, comentadas en el punto 2.4.3, permiten definir componentes como clases o funciones extendiendo de una clase que proporciona ReactJS que es React.Component [35]. Más adelante se podrá observar en un ejemplo práctico su uso.

Esta clase, React.Component junto con *react-dom*, proporciona una serie de métodos que pueden ser sobrescritos por el programador para ejecutar una funcionalidad deseada en un momento concreto del ciclo de vida del componente [35].

El ciclo de vida de un componente de ReactJS consta de dos ciclos diferentes de dos fases heterogéneas por cada uno de estos ciclos y un ciclo último con una única fase.

Estos tres ciclos son: montaje, actualización y desmontaje. Y dentro de estos existen, como ya se comentaba anteriormente, dos fases a excepción del último ciclo: fase de renderizado o fase render y fase commit. En la fase de renderizado, se inicializará y/o pintará el componente de ReactJS y en la fase de commit, el componente estará listo para su manipulación a través del DOM, actualizar el componente entre otras acciones. Las fases son comunes en Cada uno de estos estados y fases tienen una serie de **métodos propios** que permiten controlar el comportamiento y el funcionamiento del componente en dichos momentos de su ciclo de vida. En la figura 4-3 se pueden ver todos los ciclos y las fases mencionadas, así como los métodos propios de cada una de ellas.

1. **Ciclo de montaje**: En este primer ciclo, el componente se inicializa con ciertos datos que pueden venir como parámetros (más adelante se hablará de ello) o como datos (objetos, variables, valores, etc.) nuevos y creados propiamente en el componente que se está montando.

    a. **Fase de renderizado:** en esta fase se tiene a disposición varios métodos. Uno de ellos es el método ***constructor()*** el cual se encargará de inicializar el estado (se hablará del estado en el punto 4.1.2) del componente. El constructor sin embargo, no hace falta que este sea implementado; pero si se hace, se debe llamar antes que cualquier inicialización al constructor de la clase React.Component con ***super()***. Si se situara la mirada hacia un lenguaje más popular como es Java, se podría decir que, el constructor de ReactJS, es similar al de Java. En la fase de renderizado también se encuentra el método más importante de todos que es ***render()*** el cual permite incluir los componentes de ReactJS declarados dentro de este método así como componentes HTML puros dentro del DOM. Dentro del método ***render()*** no se debe modificar el estado del componente. Esto quiere decir que, cada vez que se invoque este método, debe devolver el mismo resultado [35]. En la figura 4-1 se muestra un ejemplo de lo que es la implementación de un constructor. En este caso se ha escogido el constructor de un componente de tipo combo box de localizaciones.

```
constructor(props) {
    super(props);
    this.state = { locations: this.props.data.slice(0), formError: false, errors: [] };
    if (this.state.locations.length == 0) {
        this.state.locations.push({id:''});
    }
}
```

Figura 4-1: Constructor de un componente de ReactJS. Fuente propia.

Del mismo modo la figura 4-2 muestra un ejemplo del método ***render()*** donde se puede apreciar el uso de componentes HTML y ReactJS usados junto con JavaScript. El uso de JavaScript junto con los componentes hace de ReactJS una herramienta mucho más dinámica y potente. Vemos cómo se puede embeber JavaScript mediante el uso de corchetes. Este método ***render()*** pertenece a un componente de gestión de las localizaciones de un club.

```
render = () => {
    const errors = this.state.errors;
    // Generate array of fields from each location
    let locationsFields = this.state.locations.map((location, index) =>

        <Text key = { index }
            name = { 'location' + (index + 1) }
            label = { u.t('Club {0}', (index + 1)) }
            validationType = { ValidationType.UNIQUE_TEXT }
            value = { location.id }
            onChange = { (e) => { this.onChange(e, index) } }
            removable = { this.state.locations.length > 1 }
            onRemove = { () => { this.removeLocation(location) } }
            error = { this.state.errors['location' + (index + 1)] }
            maxLength = { 200 }
            required />
    );

    return (
        <Form heading = { u.t('Localizaciones') }
            prevStep = { this.saveAndPrev }
            nextStep = { this.saveAndContinue }
            summary = { this.props.summary }
            formError = { this.state.formError }
            id = 'clubLocationsForm' >

            <p> { u.t('Introducir nombre localizacion') } </p>

            { locationsFields }

            <Button
                type = 'secondary'
                text = { u.t('Añadir otro club') }
                onClick = { this.addLocation } />
        </Form>
    );
}
```

Figura 4-2: Ejemplo de un método *render()* en ReactJS. Fuente propia.

b. **Fase *commit*:** en esta fase se podrá interactuar con el componente que se acaba de montar. En esta fase de este primer ciclo de montaje, se ofrece un método para ello llamado ***componentDidMount()*** con el que se puede volver a cambiar el estado del componente una vez renderizado. Si se quisiera cambiar este estado, el método ***render()*** se llamaría por una segunda vez de manera interna, pero ReactJS controlará este efecto de doble renderizado o estado intermedio, haciendo que sea invisible para el usuario. Este método se suele usar para cargar datos en el componente provenientes de un servidor remoto. Por lo tanto, es un buen sitio para implementar las peticiones al servidor y manejar su respuesta [35].

2. **Ciclo de actualización:** en este ciclo, el componente se actualiza tras algún cambio en los datos del mismo, es decir, en el estado del componente, bien debido a que se ha hecho uso del método ***setState()*** (explicado en el punto 4.1.2) o del método ***forceUpdate()*** los cuales llaman inmediatamente al método ***render()*** [35].

a. **Fase de renderizado:** como ya se ha dicho, tras la ejecución de uno de los dos métodos, ***setState() o forceUpdate()*** se ejecutará el método ***render()*** para actualizar el DOM del navegador con los nuevos datos del componente [35].
b. **Fase *commit:*** una vez se ha renderizado el componente se puede interactuar de nuevo con él mediante el uso del método ***componentDidUpdate()*** el cual permite al programador realizar acciones en función del estado previo a la actualización, o de la comparación entre estados (previo y nuevo tras la actualización) [35].

3. **Ciclo de desmontaje:** cuando un componente va a ser desmontado o borrado del DOM, antes de que esto ocurra, se pueden ejecutar ciertas acciones [35].

    a. **Fase *commit*:** las acciones previas al borrado y desmontado del componente se pueden manejar con el método ***componentWillUnmount()*** en el cual se podrán realizar acciones como destruir *timers* de JavaScript, o cerrar conexiones con el servidor. Sin embargo, nunca se podrá actualizar el estado del componente en este momento, es decir, no se podrá hacer uso del ***setState()*** [35].

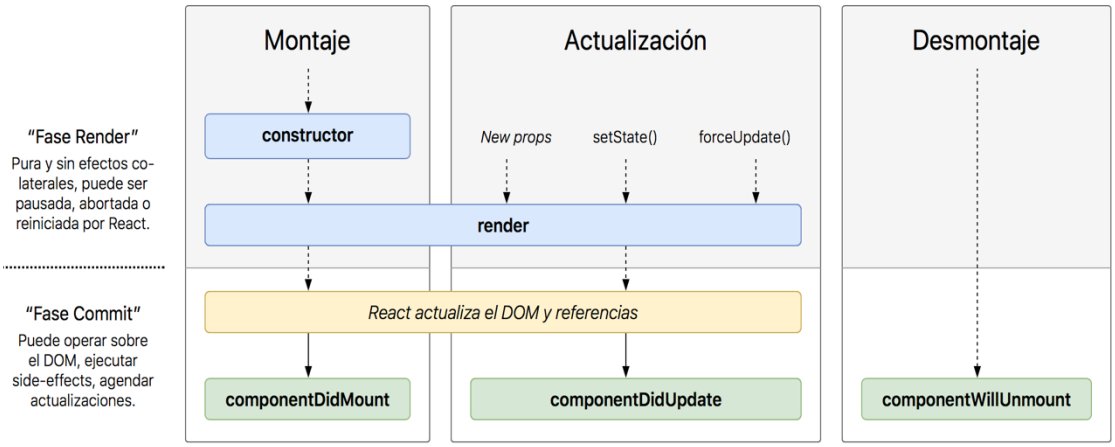

Figura 4-3: Ciclo de vida de un componente en ReactJS. Fuente http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

4.1.2 State y Props

Cualquier programador que empiece aprendiendo ReactJS se encontrará con que los componentes no son una unidad aislada y estática, sino que estos tienen "vida propia" como ya se ha ido viendo a lo largo del punto 4.1.1. Pero eso no es todo. Estos componentes de ReactJS no dejan de estar en ningún momento fuera del DOM y es por ello por lo que requieren de una vía de comunicación entre ellos. Si se imagina el DOM como una estructura arbórea, podemos decir que entre nodos padre y nodos hijo debe de existir una manera de comunicarse.

Como ya se ha dicho, un componente de ReactJS está "vivo", y esto es gracias a su estado, como se conoce en ReactJS, State [36].

Por otro lado, un componente "padre" de ReactJS se comunica con otro componente "hijo", de manera unidireccional, mediante las propiedades o ***Props***.

Se puede decir que el ***State*** guarda el conjunto de atributos que definen al componente. Ya se vio anteriormente como en el ciclo de montaje en la fase de renderizado, el constructor del componente inicializa el mismo (en el constructor no hace falta llamar a ***setState();*** para inicializar el estado). Estos atributos son usados generalmente en el método ***render();*** lo que hace que cada vez que se actualice el ***State***, se renderice todo el componente. Sin embargo, para modificar el estado hace falta ejecutar el método que ReactJS proporciona al programador. Se trata del método ***setState()*** el cual actualizará el estado y ejecutará inmediatamente el método ***render()***. Sin embargo, el uso de ***setState()*** está específicamente recomendado cuando el componente se ha renderizado, como por ejemplo, dentro del método ***componentDidMount()*** o en métodos ***callback*** tras producirse un evento, consiguiendo así una mayor eficiencia al no tener que renderizar toda la página debido al cambio de un solo componente [36].. En la figura 4-4 se puede ver un ejemplo de uso de ***setState()*** en el código. En este caso el ***State*** posee los atributos *errors* **y** *formError* entre otros.

```
saveAndContinue = () => {
    Validations.validate('clubLocationsForm').then((errors) => {
        if (Object.keys(errors).length > 0) {
            this.setState({errors, formError: true});
        } else {
            this.props.nextStep(this.state);
        }
    });
}
```

Figura 4-4: Uso de los setState() en un método callback de un componente de ReactJS. Fuente propia.

Por otro lado, se tienen los Props. Estos se definen por ser las propiedades o parámetros que contienen información para los componentes hijos que luego usarán en su método ***render()*** generalmente (aunque se pueden usar en todo el componente para otras operaciones), para mostrar dicha información o seguir pasándola hacia los componentes hijos. Es una vía de comunicación unidireccional, por lo que solo los componentes hijos pueden leer de ella, pero no escribir sobre ella para comunicar a los componentes padres información. Para comunicarse de hijos a padres se tiene que hacer uso de ***callbacks*** los cuales suelen viajar dentro de los props. En la figura 6.6 se puede apreciar el uso de los ***Props*** en ReactJS en el componente personalizado *Button*. Aquí también se puede apreciar como, aunque HTML5 en este caso proporciona un componente *button*, se crea un componente *Button* personalizado que da la opción de convertirse en un hipervínculo, texto plano o en una imagen que hará de botón; o el de ser un botón como tal. Esto se puede decidir fácilmente en los parámetros del componente ***Button*** con una llamada simple como esta: *<Button className = "clase1" href />*. Vemos como className y href son propiedades que luego el componente Button puede leer de la siguiente forma: *this.props.className* o *this props.href*. La propiedad *href* en este caso hará que el componente se comporte como un hipervínculo [37].

```
export default class Button extends React.Component {

    render = () => {

        let icon = this.props.icon ? <i className = { this.props.icon } /> : undefined;
        if (this.props.href) {
            return (
                <a
                    href = {this.props.href}
                    className = { 'btn' + ' ' + 'btn--' + this.props.type }
                    disabled = { this.props.disabled }
                    download = {this.props.download}
                    onClick = { this.props.onClick }
                    >
                    { icon }
                    { this.props.text }
                </a>
            );
        } else {
            return (
                <button
                    type = { this.props.submit ? 'submit' : 'button' }
                    className = { 'btn' + ' ' + 'btn--' + this.props.type }
                    onClick = { this.props.onClick }
                    disabled = { this.props.disabled }
                    data-toggle={this.props.toggle}
                    data-target={this.props.target}
                    data-backdrop={this.props.backdrop}
                    >
                    { icon }
                    { this.props.text }
                </button>
            );
        }
    }
}
```

Figura 4-5: Uso de los props en un método *render()*. Fuente propia.

4.2 Bootstrap

Uno de los requisitos que se han considerado a la hora de realizar el análisis del proyecto ha sido la de realizar una web usable. Para ello, no solo basta con realizar una interfaz potente en ReactJS, sino que es necesario realizar un buen diseño gráfico para todos los componentes, así como una buena distribución de los componentes en la página.

En algunos equipos de desarrollo suele haber algún diseñador gráfico que realiza la maquetación y el diseño de toda la aplicación web en este caso. Sin embargo, en otros muchos proyectos, como este, no se dispone de un diseñador gráfico para abordar tal cuestión. Es una de las razones por las que se ha decidido usar Bootstrap en este proyecto.

Bootstrap se trata de un *framework* sobre CSS que brinda al programador estilos para todo tipo de componentes como pueden ser formularios, botones, tablas o barras de navegación entre otros. Bootstrap es fácilmente acoplable a un proyecto de ReactJS. Pero, sin embargo, una de las ventajas principales que se ha visto a este *framework* es que usa diseño web *responsive*, lo que permite que la página web se adapte a pantallas grandes o pequeñas (pantallas de dispositivos móviles) sin que la estructura de la web se vea afectada. Aparte, Bootstrap trabaja con un *grid* dividido en columnas, para poder ubicar los elementos dentro de la página web de una manera mucho más cómoda [38][39]. Ya en la versión 4 de Bootstrap se hace uso de la propiedad presente en CSS3 llamada *cajas flexibles* o *flexboxes*. Básicamente, *flexbox* es un tipo de diseño en el cual los elementos se amoldan previsiblemente al tamaño de la pantalla donde se esté mostrando la web. Esto actúa en consonancia con el estándar de diseño web *responsive* [40].

4.3 BabelJS y PolyfillJS

En este proyecto se ha querido usar *BabelJS* junto con *PolyfillJS* para poder usar la última versión de JavaScript disponible y estable. *BabelJS* se trata de un compilador de JavaScript que su función principal es traducir la sintaxis de las nuevas versiones de JavaScript a una sintaxis que luego el intérprete de JavaScript del navegador pueda entender. Si bien es cierto que JavaScript está muy extendido entre los navegadores, no todos incluyen la última versión de este y es por ello por lo que, al usar *BabelJS*, se puede asegurar que la funcionalidad desarrollada se mantiene sea cual sea el entorno [41].

PolyfillJS por otro lado nos permite usar nuevas clases incluidas en las nuevas versiones de JavaScript como por ejemplo *Promise* (interesante para realizar llamadas asíncronas al servidor o alguna otra tarea asíncrona) o usar métodos como *Array.prototype.includes* (para comprobar si en un vector existe un elemento) u *Object.assign* (usado para copiar y asignar en un objeto nuevo todos los valores de todos los atributos de un objeto antiguo) [42].

Si no se hubiera usado este compilador de JavaScript, el requisito no funcional de portabilidad RNF2 no se hubieran cumplido.

4.4 Comunicación cliente-servidor

Como ya se ha ido diciendo, el proyecto consta de dos partes diferentes. La parte cliente (*front-end*) y la parte servidor (*back-end*). Este proyecto se encarga de la realización del cliente de la aplicación web, sin embargo, esta no puede existir sola. Para poder funcionar hace falta consumir servicios del servidor.

Es por ello por lo que se ha desarrollado una interfaz en JavaScript para gestionar la comunicación entre el cliente y el servidor, para que las llamadas en todo el proyecto sean sencillas y para no repetir código, lo que conlleva una mejor mantenibilidad del proyecto.

La comunicación cliente-servidor se realiza a través de llamadas a procedimientos remotos (RPC). En la figura 4-6 podemos observar cómo se realiza una llamada a un procedimiento remoto llamado *DeleteCourseById* que borrará un recorrido por su id. El funcionamiento es el siguiente. La función *rpc()* recibe un objeto JavaScript donde en la propiedad *type* se encuentra el nombre del servicio que se desea consumir seguido de otras propiedades opcionales como es *id* que contiene el id del recorrido que se desea borrar. Este *id* se utilizará de filtro para realizar una operación DELETE FROM sobre base de datos. Ya en la figura C-0-1 (ver anexo C) se puede observar como es el método *rpc()* por dentro. Su funcionamiento a grandes rasgos es simple. En primer lugar, se realiza una llamada a fetchUrl que hará la llamada al servidor. En esta llamada, se pasan como parámetros un objeto JavaScript con las cabeceras de una petición HTTP como se puede observar en la figura C-0-1 (ver anexo C). FecthUrl devuelve una promesa (Promise) JavaScript ya que la llamada al servidor es asíncrona. Esta promesa es necesaria para poder manejar la respuesta del servidor como se puede observar en las llamadas al método *then()* de Promise en la figura C-0-1 (ver anexo C), las cuales extraerán el texto de la respuesta. Dentro de *fetchUrl()* se puede observar cómo se monta la URL para después, junto con las cabeceras que ya se habían definido en un objeto JavaScript, en la llamada a *fetchUrl()*, se monta la petición con *new Request()* tal y como se puede ver en la figura C-0-2 (ver anexo C). Por último, se puede observar, en la figura C-0-2 (ver anexo C), que se devuelve una promesa JavaScript cuya función al ser ejecutada es llamar a *fetch()* (método propio de JavaScript) que realizará la petición HTTP y tratará el código de estado de la respuesta HTTP (200 si todo ha ido bien, en cualquier otro caso error).

```
remove = () => {
    DeleteDialog.show(u.t('Eliminar recorrido'), u.t('¿Está seguro que desea
        rpc({
            type: 'DeleteCourseById',
            id: this.props.course.id
        }).then(() => {
            window.location.href = '#!/administration/courses/1';
        });
    });
}
```

Figura 4-6: Llamada a procedimiento remoto. Fuente propia.

5 Pruebas y resultados

Se han realizado una serie de pruebas básicas de caja blanca y caja negra a la aplicación web para verificar su correcto funcionamiento.

1. **Pruebas de caja blanca:** debido a que las pruebas de caja blanca se definen como pruebas centradas en los detalles procedimentales del software [43], se ha probado cada componente de manera individual, así como los diferentes *callbacks* suscritos a cada evento (evento de carga, de *clic*, de *scroll*).

2. **Pruebas de caja negra:** En este tipo de pruebas se estudia un elemento desde el punto de vista de las entradas y salidas que recibe y genera respectivamente [44]. El módulo que ha sufrido mayores pruebas de caja negra ha sido el módulo de comunicación entre cliente y servidor para comprobar si realmente estaba llegando la información a los componentes que requerían de ella. Se ha planteado una prueba para medir los tiempos de carga de algunas pantallas que pueden sufrir mayor retraso.

Las pruebas de caja blanca se han realizado por medio de la herramienta Jasmine.js [45] la cual permite realizar test unitarios de una manera cómoda. Jasmine permite realizar las pruebas en ReactJS sin tener que modificar el código de manera abrupta. Se puede incluir las pruebas unitarias de la herramienta Jasmine.js en los mismos componentes de ReactJS he incluso utilizar propiedades de los propios componentes como el *State* o los *Props,* de los cuales se hablaban en la sección 4 de este documento.

A continuación, se podrá apreciar un ejemplo de prueba unitaria usando la herramienta de Jasmine.js para comprobar si el componente *Avatar,* donde se muestra la imagen del usuario y su nombre, como se puede apreciar en la figura D-1 del anexo D, se ha renderizado correctamente y también se comprueba si el nombre que contiene el componente es el que le llega por los *Props* desde el componente "padre".

```
//Prueba de renderizado para un componente en React

var props = this.props;

it("prueba de renderizado para el componente avatar", function(){
  //Se renderiza el componente con los parametros que podría recibir
  var component = renderComponent(<Avatar id = {props.id} name={props.username} />);
  //Se busca si se ha renderizado bien los componentes
  //Y si es así se busca en concreto el contenedor donde estará el nombre
  expect(component).toContainElement("div.avatarName");
  expect(component.find("div.avatarName").toContainText(props.user);
});

function renderComponent(component){
  var componente_aux = React.addons.TestUtils.renderIntoDocument(component);
  return $(componente_aux.getDOMNode());
}
```

Figura 5-1: Prueba unitaria con Jasmine.js al componete *Avatar*. Fuente propia

Una de las pruebas de tiempos se ha realizado a la pantalla de inicio de sesión tras pulsar sobre acceder. Vemos que el tiempo total ha sido de 842 milisegundos, ya que hay 337 milisegundos en los cuales el proceso estaba inactivo, lo cual es una carga bastante rápida.

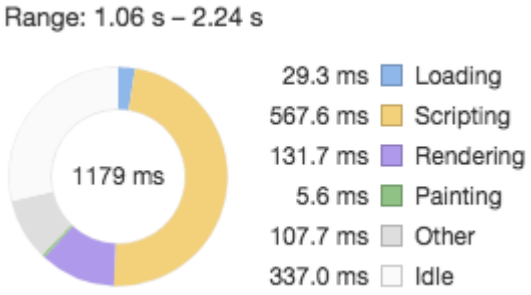

Figura 5-2: Tiempos de carga, renderizado, scripting al iniciar sesión. Fuente propia.

En esta prueba de tiempo se ha medido la carga de la pantalla de detalle de un club, puesto que tiene que cargar los detalles del club o un calendario entre otros componentes.

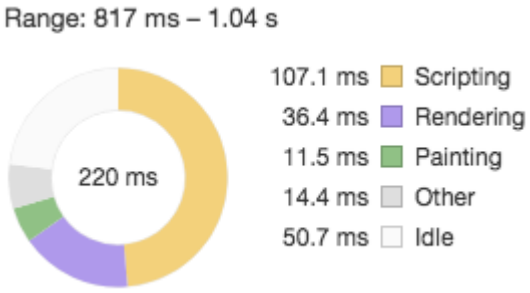

Figura 5-3: Tiempos de carga, renderizado, scripting al cargar los detalles del club. Fuente propia.

A continuación, se mostrará la breve prueba de caja negra que se ha realizado al módulo de comunicación cliente-servidor.

1. **Alta nuevo club:** En la figura 5-3 se rellenan los campos del nuevo club que se desea dar de alta y se pulsa sobre el símbolo de guardar.

Figura 5-4: Alta de nuevo club. Fuente propia.

2. **Petición HTTP con método POST:** aquí se puede observar como los campos introducidos en el formulario pasan a un objeto JSON con la información de todos los campos. Aparte, se puede observar en la figura 5-4 como en type se tiene el nombre del servicio que se va a consumir, en este caso llamado *InsertClub*.

Figura 5-5: HTTP Request Payload. Fuente propia.

3. **Respuesta del servidor:** en este caso como se trata de una operación de inserción, la respuesta será un *HTTP Status 200 OK* para informar de que todo ha ido correctamente, como se puede observar en la figura 5-5.

Figura 5-6: Respuesta HTTP del servidor. Fuente propia.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Las conclusiones a este trabajo son la posibilidad de realizar un software sencillo, modular, orientado a la web capaz de amoldarse a los requisitos que demanda cada cliente. Se ha conseguido realizar un software que no requiere de un aprendizaje intensivo y que es usable.

Debido a la magnitud de este proyecto, esta parte se ha centrado en la interfaz gráfica de usuario, consumiendo los servicios de un servidor ya implementado en la primera fase de este proyecto. Es por ello por lo que se ha aprendido a trabajar con interfaces y llamadas remotas desde entornos completamente diferentes entre sí. Se ha aprendido también a enviar objetos JSON en peticiones POST usando el protocolo HTTP con la información necesaria para consumir determinados servicios y realizar acciones concretas sobre la información almacenada en base de datos

La mejor decisión ha sido la de usar ReactJS en este proyecto. El mundo de las aplicaciones web está creciendo a pasos agigantados y es por ello por lo que no basta con poseer conocimientos de una sola tecnología. Hace falta más. La demanda de este tipo de aplicaciones es cada vez mayor y cada vez se delega más y más carga en el lado del cliente que en el servidor, debido al aumento de la potencia de procesamiento en los terminales. Es por ello por lo que las nuevas tecnologías, *frameworks* o librerías para el desarrollo de aplicaciones web facilitan el trabajo de desarrollar un producto eficiente sin por ello tener que invertir mucho tiempo. ReactJS justamente proporciona esto a este proyecto. Y no solo eso. ReactJS hace que el desarrollo sea más natural permitiendo una mejor cohesión en el código.

La aplicación al ser genérica será amoldable a cualquier club de golf por lo que una vez esté aceptada por el cliente se podrán dar de alta los diferentes clubs de golf.

Otra razón por la que haber elegido ReactJS es una decisión correcta, es por la facilidad de poder modularizar el código en pequeños componentes, como ya se veía en el punto 6, ya que una vez lanzado el producto se llevará a cabo la fase de mantenimiento de este, por lo que este hecho facilitará mucho esta tarea.

En cuanto a la usabilidad, me hubiera gustado centrarme más en ella. Pero es un tema bastante amplio al que, añadido a la complejidad del proyecto, resultaba casi imposible haber indagado más. Creo que la primera aproximación a la usabilidad esta realizada atendiendo a las características de la usabilidad que se mencionaban en el punto 3.2.1 pero no es suficiente.

El proyecto me ha parecido algo complejo debido a la cantidad de requisitos que se han impuesto desde el cliente de la aplicación. El tiempo lo ha sido todo y a pesar de haber trabajado durante mucho tiempo en esta aplicación, el resultado final aún no está alcanzado. Aparte de en un futuro poder terminarlo en la empresa, creo que el proyecto tiene un gran horizonte y puede llegar a expandirse rápidamente en todo el mundo del golf, como lo esta haciendo en estos momentos, ya que se encuentra en producción, pero en un

numero reducido de clientes. No obstante, he de decir que me ha parecido una grata experiencia, el poder gestionar un proyecto real pasando por todas sus fases desde el análisis hasta el desarrollo y pruebas. También ha sido una experiencia enriquecedora el hecho de haber podido estar de cara al cliente para la captura de requisitos o simplemente el hecho de poder comentar los avances. Por ello, creo que, aun siendo un proyecto complejo, se ha podido llevar adelante.

6.2 Trabajo futuro

A pesar de que este sistema ya se encuentra en producción a nivel privado en la intranet de unos pocos clientes siempre se pueden plantear mejoras al sistema. Estas mejoras no han sido habladas con el cliente aún, pero ya en la empresa se han planteado algunas ideas.

Para un futuro, una posible mejora sería implementar un sistema de seguimiento GPS mediante pulseras o relojes para que, desde la aplicación, en un mapa del campo de golf se pueda ver el número de personas que están en un recorrido o zona y poder enviarles información acerca de sus puntuaciones, tiempo de juego, clasificaciones del torneo, etc. Esto se podría realizar en los módulos de campos (courses) llegar a integrar Google Maps el cual leerá de base de datos todos los puntos de geolocalización de cada usuario presente en el campo.

También se ha pensado en la posibilidad de crear una aplicación móvil. ReactJS ofrece una serie de librerías para llevar la lógica de ReactJS a un dispositivo móvil de manera sencilla. No haría falta tener que invertir mucho tiempo en cambiar toda la aplicación a excepción de algunos estilos específicos para la interfaz móvil. La herramienta que ofrece ReactJS para ello es React Native. Con esta aplicación se podría llevar a cabo la mejora del seguimiento GPS. En una pantalla se pondría dos opciones posibles. Una de ellas iniciar recorrido y la otra opción posible es parar el recorrido, de manera que cuando se inicia el recorrido, la aplicación empieza a enviar a base de datos las coordenadas GPS cada en intervalos de tiempo pequeños. Inmediatamente la base de datos está siendo leída por la aplicación web.

Como se comentaba en las conclusiones, como punto futuro queda pendiente la realización de un estudio más en profundidad de la usabilidad.

Referencias

[1] "Accessibility, Usability, and Inclusion: Related Aspects of a Web for All". W3C. https://www.w3.org/WAI/intro/usable.

[2] "Usabilidad". Wikipedia. https://es.wikipedia.org/wiki/Usabilidad.

[3] "Historia del Juego del Golf". La Web de Golf. http://www.lawebdegolf.com/historia/golf.php.

[4] "Software de Gestión y Reservas para Campos de Golf". Bookgy. https://bookgy.com/es/software-gestion-reservas-online/campos-de-golf/.

[5] "Concept Golf Management Software". Concept Software Systems. http://www.csscorporate.com/en/products/concept-golf-management-software/.

[6] "Los beneficios del desarrollo de aplicaciones a medida para su empresa". CTi Soluciones. http://www.ctisoluciones.com/desarrollo-aplicaciones-a-medida/.

[7] "HTML & CSS". W3C. https://www.w3.org/standards/webdesign/htmlcss.

[8] "HTML" MDN Web Docs. https://developer.mozilla.org/es/docs/Web/HTML.

[9] "Qué es Web 2.0". Instituto Internacional Español de Marketing Digital. https://iiemd.com/web-2-0/que-es-web-2-0.

[10] "HTML 5.2" W3C Recommendation, 14 December 2017. https://www.w3.org/TR/html52/.

[11] "Cómo funciona CSS". MDN Web Docs. https://developer.mozilla.org/es/docs/Learn/CSS/Introduction_to_CSS/Como_funciona_CSS.

[12] "CSS Color Module Level 3". W3C Recommendation 07 June 2011. https://www.w3.org/TR/2011/REC-css3-color-20110607/.

[13] Varma, Vasudeva. "Software Architecture: A Case Based Approach". Delhi: Pearson Education India. p. 29. ISBN 9788131707494.

[14] "JavaScript". MDN Web Docs. https://developer.mozilla.org/es/docs/Web/JavaScript.

[15] "React – A JavaScript Library for building interfaces". React. https://reactjs.org/.

[16] "AngularJS – Superheroic JavaScript MVW Framework" AngularJS. https://angularjs.org/.

[17] "Vue.js The Progressive JavaScript Framework". Vue.js. https://vuejs.org/.

[18] "Trail: Creating a GUI With JFC/Swing". The Java Tutorials. https://docs.oracle.com/javase/tutorial/uiswing/.

[19] "[GWT] Overview". GWT. http://www.gwtproject.org/overview.html.

[20] "Entendiendo qué es GWT". Blog Tecnología para Desarrollo. https://www.paradigmadigital.com/dev/entendiendo-que-es-gwt/.

[21] "PHP General Information - Manual" PHP. http://us.php.net/manual/en/faq.general.php.

[22] "PHP". Wikipedia. https://es.wikipedia.org/wiki/PHP.

[23] "HHS.gov Web Standards" U.S. Department of Health & Human Services. https://webstandards.hhs.gov/guidelines/.

[24] "¿Qué es la usabilidad?" Guía Digital Gobierno de Chile. http://www.guiadigital.gob.cl/index.html.

[25] "Estándares – W3C España". W3C España. https://www.w3c.es/estandares/.

[26] "Requisito funcional". Wikipedia. https://es.wikipedia.org/wiki/Requisito_funcional.

[27] "Requisito no funcional". Wikipedia. https://es.wikipedia.org/wiki/Requisito_no_funcional.
[28] "The Single Page Interface Manifesto". The Single Page Interface Manifesto. http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php.
[29] "Single-page application". Wikipedia. https://es.wikipedia.org/wiki/Single-page_application.
[30] "Introduction to XML". W3Schools. https://www.w3schools.com/xml/xml_whatis.asp
[31] "JSON Introduction". W3Schools. https://www.w3schools.com/js/js_json_intro.asp.
[32] International Organization for Standadization (ISO), Ergonomics of human-system interaction. Guidance on Usability. ISO 9241-11:1998.
[33] "Estándares formales de usabilidad y su aplicación práctica en una evaluación heurística." Usable Accesible. https://olgacarreras.blogspot.com/2012/03/estandares-formales-de-usabilidad-y-su.html.
[34] "Document Object Model". Wikipedia. https://es.wikipedia.org/wiki/Document_Object_Model.
[35] "React Component". React. https://reactjs.org/docs/react-component.html.
[36] "State and Lifecycle". React. https://reactjs.org/docs/state-and-lifecycle.html.
[37] "Components and Props". React. https://reactjs.org/docs/components-and-props.html.
[38] "Introduction Bootstrap". Bootstrap. https://getbootstrap.com/docs/4.0/getting-started/introduction/
[39] "Bootstrap 4 Get Started". W3Schools. https://www.w3schools.com/bootstrap4/bootstrap_get_started.asp
[40] "Usando las cajas flexibles CSS". MDN Web Docs. https://developer.mozilla.org/es/docs/Web/CSS/CSS_Flexible_Box_Layout/Usando_las_cajas_flexibles_CSS.
[41] "What is Babel?" Babel. https://babeljs.io/docs/en.
[42] "babel-polyfill". Babel. https://babeljs.io/docs/en/babel-polyfill/.
[43] "Pruebas de caja blanca". Wikipedia. https://es.wikipedia.org/wiki/Pruebas_de_caja_blanca
[44] "Caja negra (sistemas)". Wikipedia. https://es.wikipedia.org/wiki/Caja_negra_(sistemas).
[45] "Getting Started". Jasmine Behavior-Driven JavaScript. https://jasmine.github.io/pages/getting_started.html

Glosario

TIC	Tecnologías de la Información y Comunicación.
Callback	Función A que se utiliza como argumento de otra función B y solo se ejecuta cuando se ejecuta B.
RPC	Llamada a procedimiento remoto
Front-end	En una aplicación, la capa de presentación.
Back-end	En una aplicación, la capa de servicios y acceso a datos.
Grid	Cuadrícula, malla donde se distribuyen elementos.
Responsive	Diseño web adaptativo a cualquier dispositivo.
Tee time	Nombre que recibe la hora reservada para jugar. Normalmente cada 8 minutos.
Green Fee	Es el cargo por jugar una ronda o sesión en un campo de golf.
Courses	Campos, recorridos.
Scroll	Desplazamiento en la pantalla horizontal o vertical.
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
DOM	*Document Object Model.*
Timer	Elemento que lanza un evento de reloj en determinados periodos de tiempo.

Anexos

A Manual del programador e instalación

En este proyecto en concreto se han hecho uso de varias tecnologías para el despliegue y la configuración del proyecto. Las herramientas necesarias para ello son Node.js y Express.js proporcionado por Node.js. Node.js permitirá gestionar todas las dependencias del proyecto y desplegar el mismo, junto con la ayuda de Express.js, en un servidor llamado hot-server, que permitirá realizar cambios en caliente en el código y mostrarlos inmediatamente en el navegador. Express.js, aparte de ayudar al despliegue, brinda la posibilidad de poder configurar el servidor que despliega junto a Node.js. Algunas otras herramientas que se han usado son Babel que junto con Webpack.js (fichero de configuración donde se cargarán herramientas como Babel para la traducción de ReactJS a JavaScript y cuyo principal objetivo es empaquetar todas las dependencias, código ReactJS, imágenes entre otras cosas y convertirlos en estáticos y unificarlo).

Antes de nada, hay que instalar Node.js en el sistema. Para ello simplemente se abre una terminal y se escribe el comando ***npm install***.

1. **Inicializar el proyecto:** Se crea una carpeta nueva donde irá el proyecto. Dentro de la carpeta se procede a inicializar el proyecto con el comando ***npm init --yes***. Esto generará un fichero muy importante en el proyecto llamado ***package.json*** (figura A-3) en el cual se encuentran todas las dependencias del proyecto entre otras cosas. Para instalar las dependencias de ReactJS necesarias para el proyecto (***react*** y ***react-dom***) basta con ejecutar: ***npm install react react-dom --save***. También se desea que el proyecto soporte las últimas versiones de JavaScript. Para ello se instalan las dependencias de Babel con el comando: ***npm i babel-core babel-preset-es2015 babel-preset-react babel-loader -D***. Como los programas o componentes en ReactJS están escritos en ficheros ***.jsx***, Babel se encargará de realizar el papel de compilador entre .jsx y JavaScript. Pero no solo basta con ello, sino que también hace falta que Babel se cargue en el proyecto y configurarlo para que busque los ficheros ***.jsx*** del proyecto para que los traduzca a JavaScript. Esto se consigue mediante Webpack.js, como ya se ha mencionado anteriormente. La configuración de ***Webpack.js*** es sencilla. Se debe declarar dónde está el punto de entrada al proyecto, desde el cual empieza a aplicar las acciones que se le especifiquen, en este caso, la acción de traducir de ReactJS a JavaScript. ***Webpack.js*** se declara como dependencia del proyecto en el ***package.json*** con lo cual Node.js se encargará de obtenerlo. En la figura A-1 se puede observar cómo se invoca en ***Webpack.js*** la acción de traducir de ReactJS a JavaScript mediante Babel.

```
var loaders = [
    {test: /\.jsx(\?.*)?$/, loader: 'babel-loader?' + babelOptsJson},
```

Figura A-1: Acción de búsqueda de cualquier archivo .jsx para su posterior traducción a JavaScript con Babel.js. Fuente propia.

En la figura A-2, se puede observar cómo es la estructura del fichero de configuración webpack.config.js. En primer lugar, se observa un campo *entry*

donde se declara el punto de partida del proyecto, como se comentaba anteriormente. También se puede ver como las acciones o *loaders* descritas en la figura A-1 se instancian en este momento. Por otro lado, *filename* va a ser donde se declare la ruta del fichero donde irá traducido y comprimido en un solo fichero JavaScript todo el programa en ReactJS.

```
{
    entry: path.join(options.mainFolder, options.mainJsx),
    output: {
        path: options.path,
        filename: '[name].js' + (options.longTermCaching && !options.prerender ? '?hash=[chunkhash]' : ''),
        chunkFilename: (options.devServer ? '[id].js' : '[name].js') + (options.longTermCaching && !options.p
        sourceMapFilename: 'debugging/[file].map',
        libraryTarget: options.prerender ? 'commonjs2' : undefined,
        pathinfo: !options.minimize, // Para que en los generados se incluya un
                                     // comentario con la ruta de cada fichero
    },
    target: options.prerender ? 'node' : 'web',
    module: {
        loaders: loaders
    },
```

Figura A-2: Fichero webpack.config.js. Fuente propia.

```
{
    "mame": "deloniareporting",
    "version": "0.1.0",
    "private": true,
    "scripts": {
        "dev-server": "webpack-dev-server --content-base build/public --config config/webpack-dev-server.config.js --progres
        "hot-dev-server": "webpack-dev-server --content-base build/public --config config/webpack-hot-dev-server.config.js -
        "build": "webpack --config config/webpack-production.config.js --progress --profile --colors",
        "build-i18n": "webpack --config config/webpack-production-i18n.config.js --progress --profile --colors",
        "build-dev": "webpack --config config/webpack-development.config.js --progress --profile --colors",
        "build-dev-i18n": "webpack --config config/webpack-development-i18n.config.js --progress --profile --colors",
        "start-dev": "node config/server",
        "start": "node config/server",
        "clean": "rm -Rf build",
        "clean-all": "rm -Rf build node_modules",
        "npm-check-updates": "npm-check-updates"
    },
    "dependencies": {
        "react": "15.4.1",
        "react-dom": "15.4.1",
        "normalize.css": "5.0.0",
        "font-awesome": "4.7.0",
        "classnames": "2.2.5",
        "route-parser": "0.0.5",
        "intl-messageformat": "1.3.0",

        "intl": "1.2.5",
        "babel-polyfill": "6.20.0",
        "whatwg-fetch": "2.0.1",

        "webpack": "1.14.0",
        "webpack-dev-server": "1.16.2",

        "react-proxy-loader": "0.3.4",
        "null-loader": "0.1.1",
        "html-loader": "0.4.4",
        "json-loader": "0.5.4",
```

Figura A.3: Fragmento del fichero package.json. Fuente propia.

2. **Instalar express.js:** Una vez inicializado el proyecto, se procede a descargar e instalar Express.js ejecutando el siguiente comando por la terminal: ***npm i express --save***.

3. **Desarrollo y configuración del servidor hot-server:** A continuación, se debe crear dentro de la carpeta del proyecto un fichero llamado ***server.js*** el cual llamará a ***express.js*** y configurará el puerto donde estará escuchando el servidor *hot-*

server. Además, es en este fichero donde se especifica dónde está el archivo ***index.html*** que será el punto de inicio de toda la aplicación. En la figura A-4 se puede ver la implementación del ***server.js*** con ayuda de ***express.js***. Se puede apreciar en la figura como *express.static()* hace referencia a la carpeta *public* donde se encuentra ficheros estáticos del proyecto, entre ellos, el fichero *index.html* comentado anteriormente. Para probar que todo ha ido correctamente, es tan simple como ejecutar el fichero *server.js* con *Node* de la siguiente manera desde la terminal: ***node server.js***.

```
var express = require('express');
var path = require('path');
var fs = require('fs');
var port = +(process.env.PORT || 3000);
var app = express();

app.use(function(req, res, next) {
    res.setHeader('Access-Control-Allow-Origin', '*');
    return next();
});
app.use(function(req, res, next) {
    if (req.path === '/' || req.path === '/index.html') {
        var indexContent = fs.readFileSync(path.join(__dirname, '..', 'build', 'public', 'index.html'), 'utf8');
        indexContent = indexContent.replace(/\{\[\{\/script\}\]\}/g, "</script" + ">");
        indexContent = indexContent.replace(/\{\[\{protocolAndHostName\}\]\}/g, req.protocol + '://' + req.hostname);
        indexContent = indexContent.replace(/\{\[\{polyfill\}\]\}/g, getPoyfill(req.headers['user-agent']));
        indexContent = indexContent.replace(/\{\[\{locale\}\]\}/g, 'es-ES');
        indexContent = indexContent.replace(/\{\[\{serverUrl\}\]\}/g, req.protocol + '://' + req.hostname + ':' + port + '/');
        res.send(indexContent);
        return;
    }
    next();
});
app.use('/', express.static(path.join(__dirname, '..', 'build', 'public')));
app.use('/staticWebContent', express.static(path.join(__dirname, '..', 'config', 'staticWebContent')));

app.listen(port, function() {
    console.log('Server listening on port ' + port);
});
```

Figura A-4: Fragmento del fichero server.js. Fuente propia.

4. **Index.html e index.jsx:** Como ya se comentaba anteriormente, el fichero ***index.html*** es el que va a intentar buscar el servidor de *express.js*. En él es muy importante que haya un componente *div* de HTML con un id llamado *app* (si bien es cierto que se puede poner indiferentemente cualquier nombre siempre que el programador sea consecuente con ello, aunque por convención se pone el nombre propuesto). El primer componente, pero uno de los más importantes de la aplicación es ***index.jsx***. Este fichero renderizará toda la aplicación en el componente *div* con id *app* nombrado anteriormente. La manera de hacerlo queda representada en la figura A-5. Se puede ver como dentro del método *render()* se instancia el componente "padre" de toda la aplicación llamado *App* en el elemento de HTML con id *app*.

```
import React, {Component} from 'react';
import ReactDOM from 'react-dom';
import App from './app.jsx';

ReactDOM.render(<App/>, document.getElementById('app'), Loading.hide);
```

Figura A-5: Fichero index.jsx. Fuente propia.

5. **App.jsx:** Este fichero ya contiene a los primeros componentes de la aplicación web en ReactJS. A partir de este instante ya se pueden crear nuevos componentes, e instanciarlos dentro de otros componentes, partiendo siempre de que el componente *App* es el componente padre de toda la aplicación. En la figura A-6 se puede observar la implementación de *App.jsx*.

```
export default class App extends React.Component {
    onPathChanged = (path) => {
        this.menu.setCurrentPath(path);
    }
    logout = () => {
        ConfirmDialog.show('Cerrar sesión', <span>{u.t('¿Está seguro que desea cerrar sesión?')}</span>).then(() => {
            window.location.href = 'logout';
        });
    }
    render() {
        return (
            <u.horizontal className='app'>
                <u.horizontal className='u-w1'>
                    <Menu ref={this.setMenuComponnet}/>

                    <Router style={{paddingTop: '0.8rem'}} className='u-vertical u-w1' routes={routes} onPathChanged={this.
                </u.horizontal>
            </u.horizontal>
        );
    }
    setMenuComponnet = (menu) => {
        this.menu = menu;
    }
}
```

Figura A-6: Fichero App.jsx. Fuente propia.

6. **Arrancar la aplicación web:** Para arrancar el servidor *express.js* que aloja a la aplicación web, simplemente se escribe el comando: ***npm run start***. Para arrancar el servidor en modo *hot-server* para ver los cambios que se realizan inmediatamente en el navegador, se escribe: ***npm run hot-dev-server***.

B Diagramas de flujo de la aplicación.

Se van a mostrar los diagramas de flujo más complejos del sistema. Estos diagramas de flujo son compartidos con la primera parte del producto final (back-end) y con esta segunda parte (front-end). Estos están desarrollados y revisados junto con el cliente.

B.1 Diagrama de flujo nueva reserva a petición del cliente

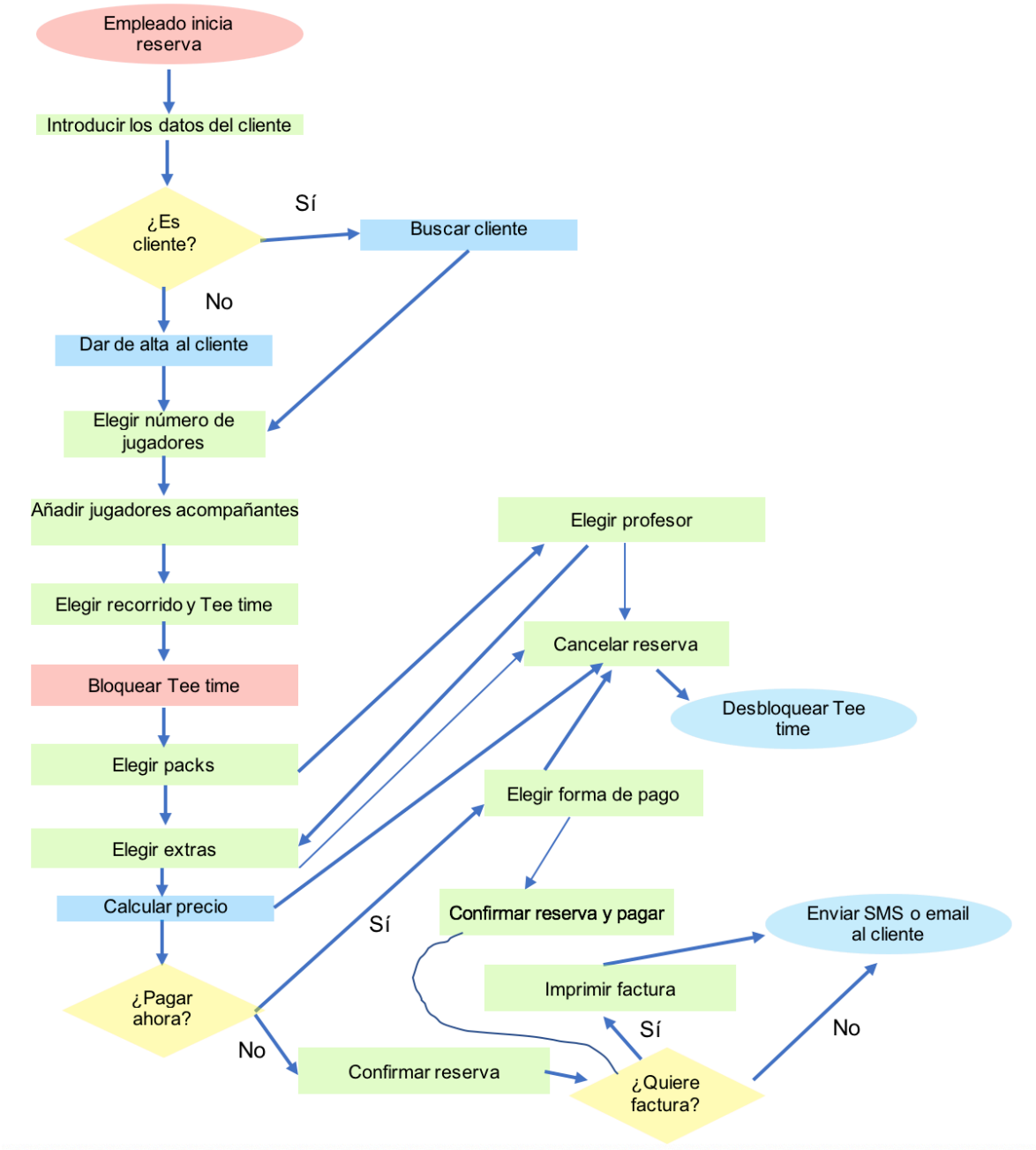

Figura B-1: Diagrama de flujo de nueva reserva a petición del cliente. Fuente propia.

B.2 Diagrama de flujo cancelación de reserva

Proceso de cancelación de una reserva realizada con anterioridad.

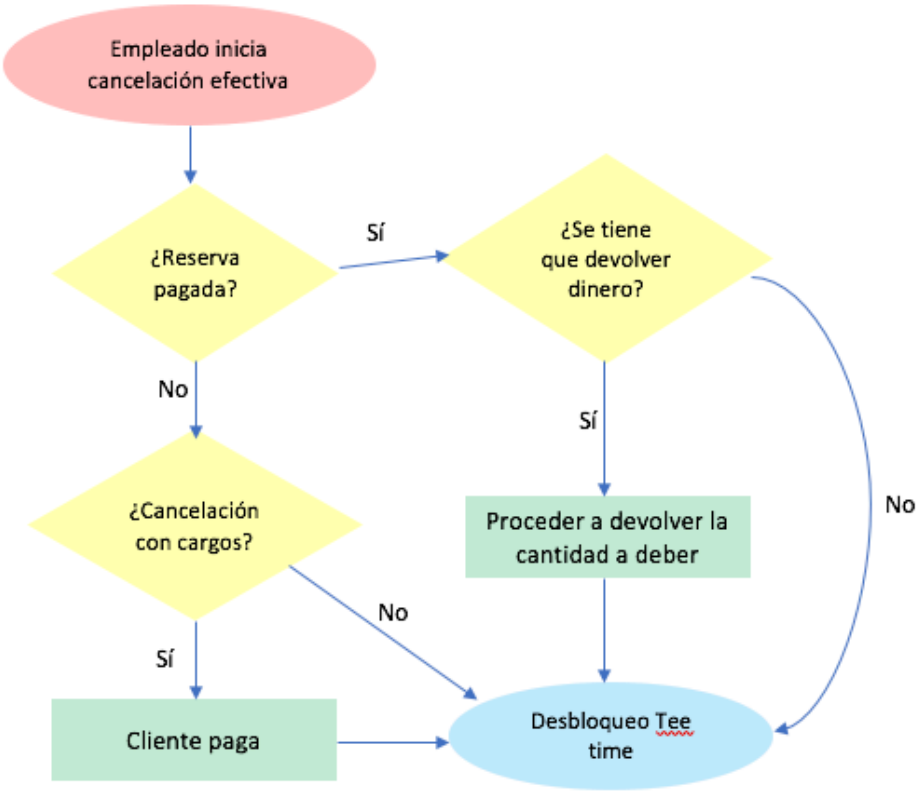

Figura B-2: Diagrama de flujo cancelación de reserva. Fuente propia.

B.3 Diagrama de flujo de gestión de la compra de un abono

Acciones realizadas por el empleado cuando un nuevo cliente o un socio quiere comprar un abono para usar el campo.

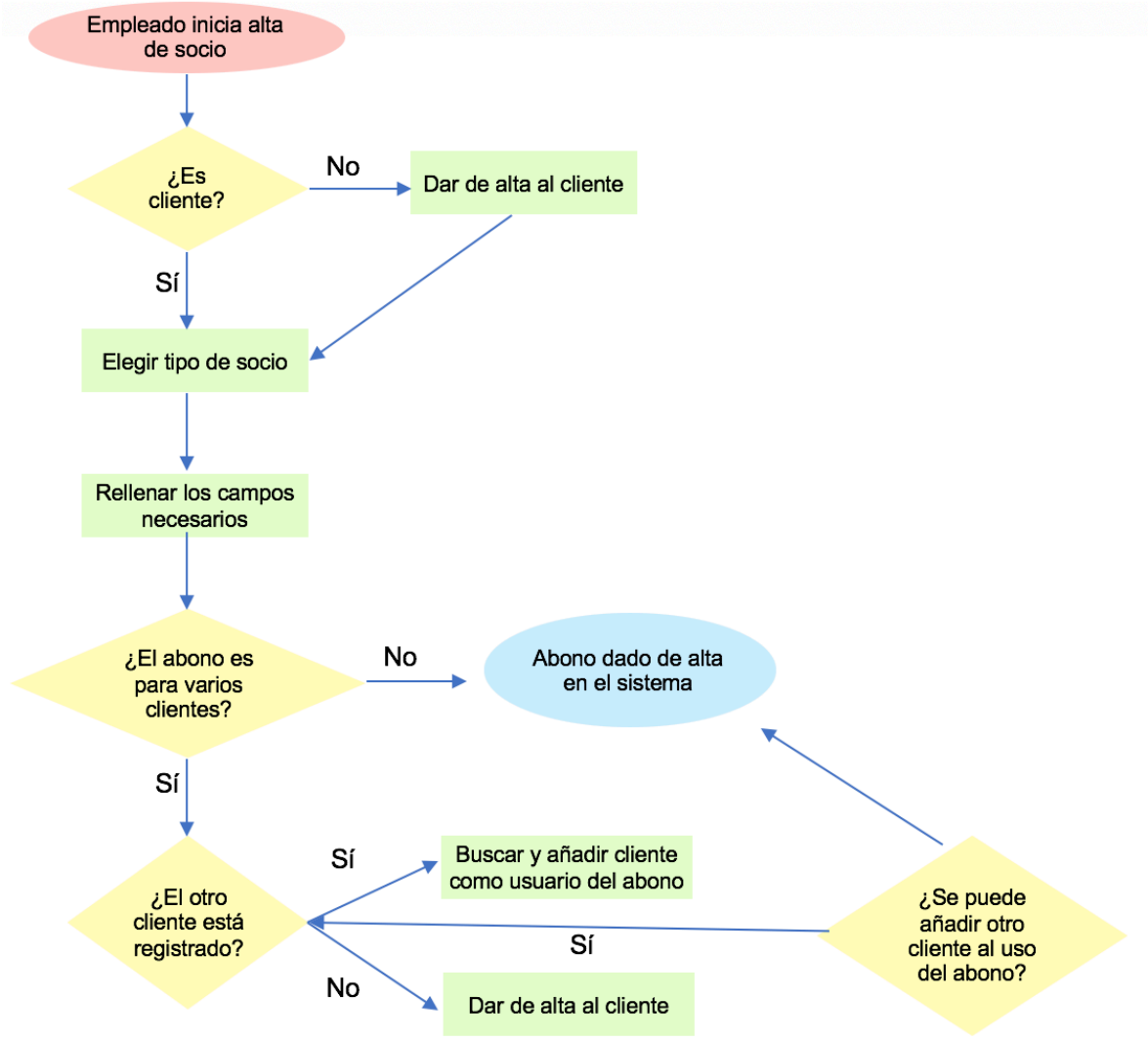

Figura B-3: Diagrama de flujo de gestión de compra de un abono. Fuente propia.

C Comunicación cliente-servidor

En este anexo se verá una breve explicación de la interfaz RPC escrita en JavaScript para el consumo de servicios del servidor web.

```
const validate = require('./validations.jsx');
const events = require('./events.jsx');

function rpc(operation) {
    if (process.env.NODE_ENV !== 'production') {
        validateOperationRequest(operation);
    }
    const result = fetchUrl('JsonService', {
        method: 'POST',
        body: toJson(operation),
        cache: 'no-cache',
        headers: {
            'Content-Type': 'application/json;charset=utf-8',
            Accept: 'application/json',
            'Cache-Control': 'no-cache'
        }
    }).then((response) => {
        return response.text();
    }).then((responseText) => {
        const opResult = fromJson(responseText);
        if (process.env.NODE_ENV !== 'production') {
            validateOperationResponse(operation, opResult);
        }
        events.emit('our.rpc.received', opResult, operation);
        return opResult;
    }, (error) => {
        events.emit('our.rpc.receivedError', error, operation);
        throw error;
    });
    events.emit('our.rpc.sent', operation);
    return result;
}
```

Figura C-0-1: Implementación del método rpc(). Fuente propia.

```
function fetchUrl(url, config = {}) {
    let finalUrl = url;
    const isRelativeUrl = !(/^(\/\/|http:|https:).*/.test(url));
    if (isRelativeUrl) {
        if (window._ourServerUrl) { // eslint-disable-line no-underscore-dangle
            finalUrl = window._ourServerUrl + url; // eslint-disable-line no-underscore-dangle
        }
        if (!config.credentials) {
            config.credentials = 'same-origin';
        }
    }

    const request = new Request(finalUrl, config);

    if (isRelativeUrl) {
        // Only add extra headers for relative url
        const headers = request.headers;
        if (!headers.has('X-CURRENT')) {
            headers.set('X-CURRENT', '@' + btoa(encodeURI(window.location.href)) + '!');
        }
        if (!headers.has('X-XSRF-TOKEN')) {
            const xsrfCookieValue = getCookie('XSRF-TOKEN');
            if  (xsrfCookieValue) {
                headers.set('X-XSRF-TOKEN', xsrfCookieValue);
            }
        }
    }

    const result = fetch(request).then((response) => {
        // status "0" to handle local files fetching (e.g. Cordova/Phonegap etc.)
        if (response.status === 200 || response.status === 0) {
            events.emit('our.rpc.fetched', url, config, response);
            return response;
        }
        const error = new TypeError('Fail executing a RPC request: unaceptable response code ' +
        error.response = response;
        events.emit('our.rpc.fetchedError', url, config, error);
        throw error;
    }, (error) => {
        events.emit('our.rpc.fetchedError', url, config, error);
        throw error;
    });
    events.emit('our.rpc.fetch', url, config);
    return result;
}
```

Figura C-0-2: Implementación del método fetchUrl(). Fuente propia.

D Capturas de la aplicación

A continuación, se muestran algunas pantallas de la aplicación web.

1. **Pantalla de listado de clientes:** se ha incluido un botón de nueva reserva en el menú lateral fijo ya que es uno de los procesos que más se repiten de la aplicación y se desea que el usuario tenga un acceso inmediato en el caso de necesitarlo para minimizar los tiempos de gestión.

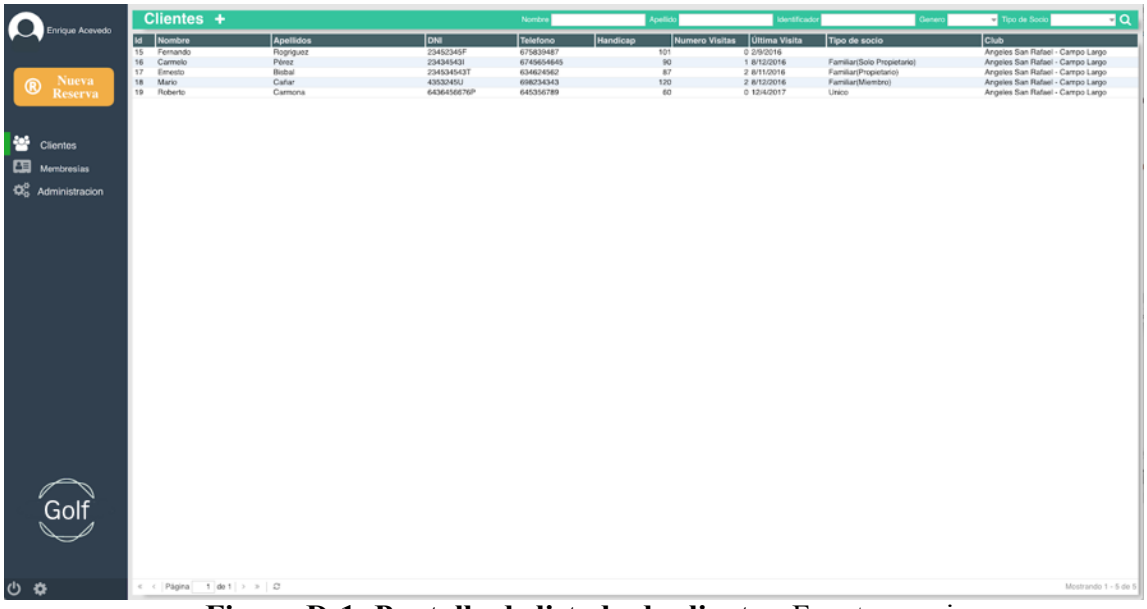

Figura D-1: Pantalla de listado de clientes. Fuente propia.

2. **Pantalla de detalle de cliente:** para mayor claridad para el usuario las alertas se han puesto en el lateral derecho superior donde los colores indican la importancia que se le ha dado a la alerta. Al hacer clic sobre ella se abrirá un *pop-up* con más información sobre la misma y a su vez se podrá marcar como tratada o que no vuelva a aparecer.

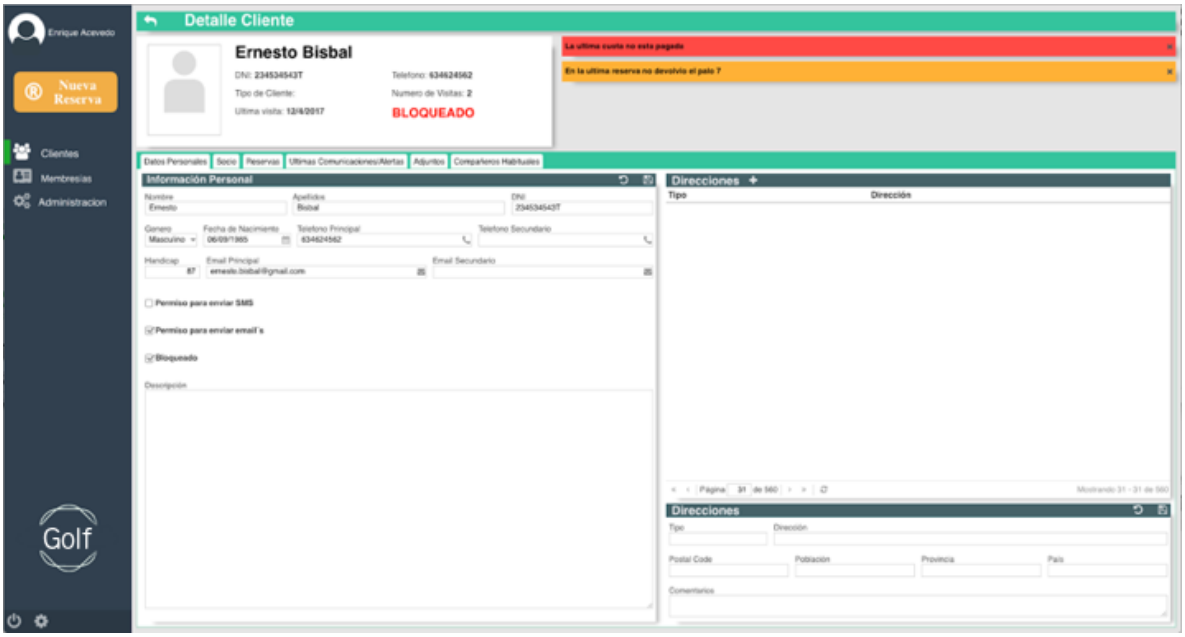

Figura D-2: Pantalla de detalle de clientes. Fuente propia.

3. **Pantalla de socio con varias membresías:** pantalla para un socio que posee varias membresías.

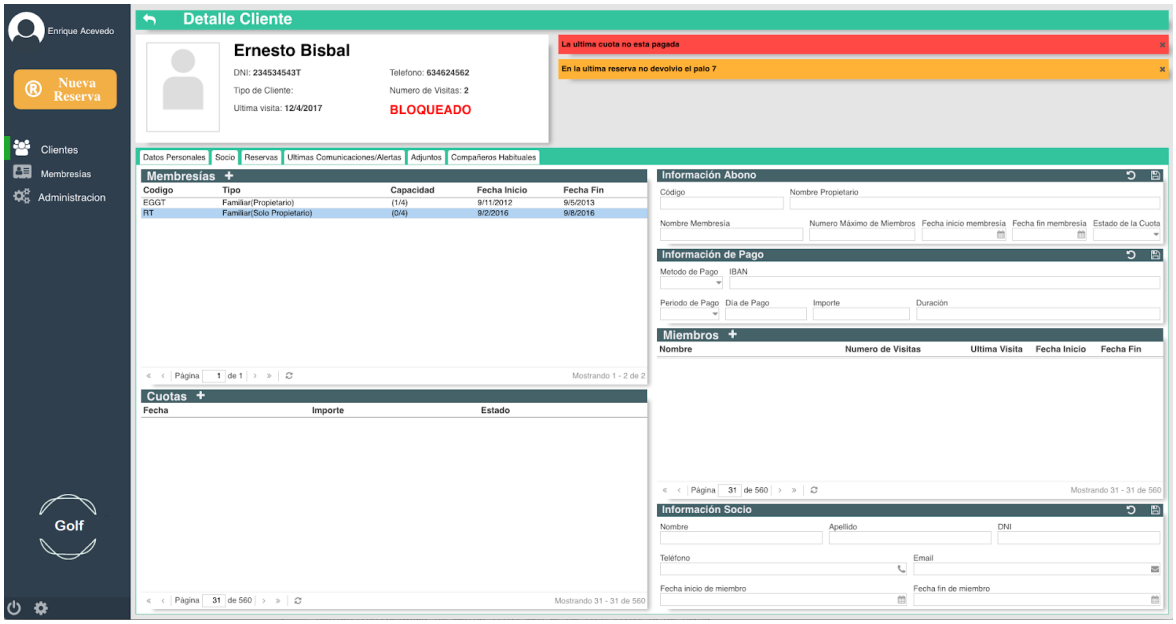

Figura D-3: Pantalla de socio con varias membresías. Fuente propia.

4. **Pantalla de reservas de un cliente:** en esta pantalla se recogen todas las posibles reservas de un cliente.

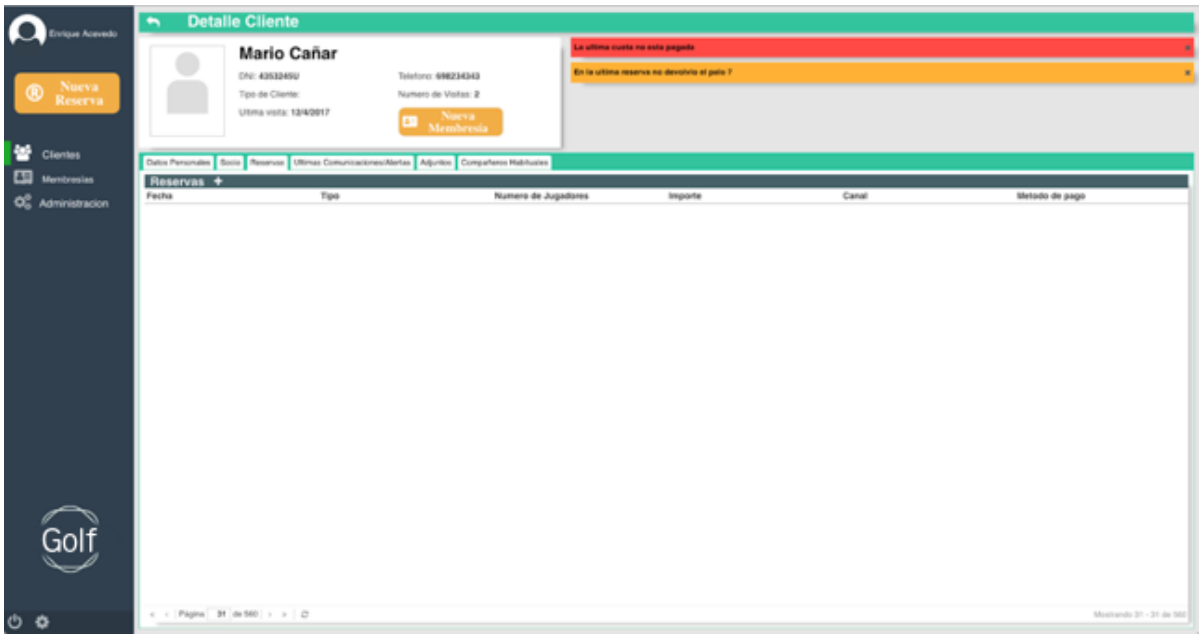

Figura D-4: Pantalla de detalles de cliente con varias membresías.
Fuente propia.